Assembler Language Programming for IBM System z™ Servers

Version 2.00

John R. Ehrman

IBM Silicon Valley Lab

Second Edition (February 2016)

IBM welcomes your comments. Please address them to


```
John Ehrman
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA 95141
ehrman@us.ibm.com
```


© **Copyright IBM Corporation 2015**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures xvi

Tables xxix

Foreword 1
Outline and Overview 1
Programming Environments 2
Levels of Difficulty (*) 2
Exercises and Programming Problems 2
Some Personal Observations 2
Von Neumann Architecture 5
Why Program in Assembler Language (and Why Not)? 5
Assembler Language Misconceptions 8

Chapter I: Getting Started 11
1. Some Basic Items 12
1.1. Notation and Terminology 12
1.2. Instruction Elements 13
1.2.1. Register Names 14
2. Binary and Hexadecimal Numbers 16
2.1. Positional Notation and Binary Numbers 16
2.2. Hexadecimal Numbers 17
2.3. Converting Integers from One Base to Another (*) 19
2.4. Examples of General Conversions (*) 22
2.5. Number Representations 24
2.6. Logical (Unsigned) Representation 25
2.7. Two's Complement (Signed) Representation (*) 25
2.8. Computing Two's Complements 27
2.9. Sign Extension 30
2.10. Binary Addition 31
2.11. Binary Subtraction 32
2.12. How Additions and Subtractions Are Actually Performed (*) 34
2.13. A Circular View of Binary Arithmetic (*) 36
2.14. Logical (Unsigned) and Arithmetic (Signed) Results (*) 37
2.15. Examples of Representations (*) 38

Chapter II: System z 41
3. Conceptual Structure of System z 42
3.1. Memory Organization 43
3.2. Central Processing Unit 45
3.3. General Registers 45
3.4. Floating-Point Registers 46
3.5. Program Status Word (PSW) 47
3.6. Other Registers 48
3.7. Input-Output (I/O) 48
3.8. Features, Facilities, and Assists 48
3.9. Microprograms and Millicode (*) 48
4. Instruction Execution 50
4.1. Basic Instruction Cycle 50
4.2. Basic Instruction Types 51
4.3. Instruction Lengths 53
4.4. Some Operation Codes (*) 54
4.5. Interruptions (*) 55
4.6. Exceptions and Program Interruptions (*) 56
4.7. Machine Language and Assembler Language 58
4.8. Processor Evolution 59
5. Memory Addressing 61
5.1. The Addressing Halfword 62

5.2. Examples of Effective Addresses . . . 63
5.3. Indexing . . . 63
5.4. Examples of Indexing . . . 65
5.5. Addressing Problems (*) . . . 66
5.6. Address Translation and Virtual Memory (*) . . . 67
5.7. Summary . . . 68

Chapter III: Assembler Language Programs . . . 71
6. Assembler Language . . . 72
6.1. Processing Your Program . . . 72
6.1.1. Assembly . . . 72
6.1.2. Linking . . . 73
6.1.3. Loading and Execution . . . 73
6.2. Preparing Assembler Language Statements . . . 74
6.3. Statement Fields . . . 76
6.3.1. What's in a Name Field? (*) . . . 79
6.4. Writing Programs . . . 79
6.5. A Sample Program . . . 80
6.6. Basic Macro Instructions . . . 82
6.7. Summary . . . 82
7. Self-Defining Terms and Symbols . . . 85
7.1. Self-Defining Terms . . . 85
7.2. EBCDIC Character Representation . . . 87
7.3. Symbols and Attributes . . . 89
7.4. Program Relocatability . . . 91
7.5. The Location Counter . . . 92
7.6. Assigning Values to Symbols . . . 93
7.7. Symbols and Variables . . . 94
8. Terms, Operators, Expressions, and Operands . . . 96
8.1. Terms and Operators . . . 96
8.2. Expressions . . . 97
8.3. Evaluating Assembly-Time Expressions (*) . . . 98
8.4. Examples . . . 100
8.5. Machine Instruction Statement Operand Formats . . . 102
8.6. Details of Expression Evaluation (*) . . . 103
9. Instructions, Mnemonics, and Operands . . . 106
9.1. Basic RR-Type Instructions . . . 106
9.2. Writing RR-Type Instructions . . . 107
9.3. Basic RX-Type Instructions . . . 108
9.4. Writing RX-Type Instructions . . . 108
9.5. Explicit and Implied Addresses . . . 109
9.6. Typical RS- and SI-Type Instructions . . . 111
9.7. Writing RS- and SI-Type Instructions . . . 111
9.8. Typical SS-Type Instructions . . . 113
9.9. Writing SS-Type Instructions . . . 113
9.10. Summary . . . 115
10. Establishing and Maintaining Addressability . . . 116
10.1. The BASR Instruction . . . 116
10.2. Computing Displacements . . . 117
10.3. Explicit Base and Displacement . . . 119
10.4. The USING Assembler Instruction and Implied Addresses . . . 120
10.5. Location Counter Reference . . . 121
10.6. Destroying Base Registers . . . 122
10.7. Calculating Displacements: the Assembly Process, Pass One . . . 123
10.8. Calculating Displacements: the Assembly Process, Pass Two . . . 125
10.9. Multiple USING Table Entries . . . 127
10.10. The DROP Assembler Instruction . . . 128
10.11. Addressability Errors . . . 129
10.12. Resolutions With Register Zero (*) . . . 130
10.13. Summary . . . 132
10.13.1. How the Assembler Helps . . . 133

Chapter IV: Defining Constants and Storage Areas **135**
11. Defining Constants 136
11.1. Defining Constants 137
11.2. DC Instruction Statements and Operands 138
11.2.1. Blanks in Nominal Values 138
11.3. Boundary Alignment 139
11.4. Length Modifiers 140
11.5. Duplication Factors and Multiple Operands 141
11.6. Multiple Nominal Values 142
11.7. Length Attributes 143
11.8. Decimal Exponents (*) 143
11.8.1. Decimal Exponents 143
11.8.2. Exponent Modifiers 144
12. Basic Constants 146
12.1. F-Type and H-Type Constants 146
12.2. A-Type Address Constants 147
12.3. Y-Type Address Constants 149
12.4. Constants of Types C, X, and B 150
12.5. Padding and Truncation 152
12.6. Literals 154
12.7. The LTORG Assembler Instruction 156
12.8. Type Extensions 157
13. Data Storage Definition 159
13.1. Storage Areas: The DS Assembler Instruction 159
13.2. Zero Duplication Factor 160
13.3. The EQU Assembler Instruction 162
13.4. EQU Instruction Extended Syntax (*) 166
13.5. The ORG Assembler Instruction 167
13.6. Parameterization 169
13.7. Constants Depending on the Location Counter 171
13.8. Assembly Time and Execution Time, Revisited (*) 173
13.9. Summary Observations 174

Chapter V: Basic Instructions **177**
14. General Register Data Transmission 178
14.1. Load and Store Instructions 179
14.2. Multiple Loads and Stores 180
14.3. Halfword Data 182
14.4. Insert and Store Character 184
14.5. ICM and STCM Instructions 185
14.6. RR-Type Data Transmission Instructions 187
14.7. Load, Store, and Insert for 64-bit General Registers 189
14.8. RRE-Type Data Transmission Instructions for 64-bit General Registers . . . 192
14.9. The Load and Test Instructions 193
14.10. Mixed 32- and 64-bit Operands 194
14.11. Other General Register Load Instructions (*) 195
14.11.1. Load Byte Instructions 196
14.11.2. Load Logical Character Instructions 196
14.11.3. Load Logical Halfword Instructions 197
14.11.4. Load Logical (Word) Instructions 197
14.11.5. Load Logical Thirty One Bit Instructions 197
14.12. Misunderstandings to Avoid 198
14.13. Summary 199
15. Testing the Condition Code: Conditional Branching 204
15.1. The Branch Address 204
15.2. The Branch Mask and Branch Condition 205
15.3. Examples of Conditional Branch Instructions 206
15.4. No-Operation Instructions 206
15.4.1. Special No-Operation Instructions (*) 206
15.5. Conditional No-Operation 207
15.6. Extended Mnemonics 210
15.7. A Comment on Programming Style 212

15.8. A Design Oversight and a Modern "Correction" (*) 212
15.9. Summary 213
16. Fixed-Point Binary Addition, Subtraction, and Comparison 216
16.1. Signed-Arithmetic Add and Subtract Instructions 216
16.2. Signed-Arithmetic Operations Using 32-Bit Registers 217
16.2.1. Condition Code Settings After Arithmetic 218
16.3. Signed-Arithmetic Operations Using 64-Bit Registers 221
16.4. Signed-Arithmetic Compare Instructions 222
16.5. Logical-Arithmetic Add and Subtract Instructions 224
16.6. Add With Carry, Subtract With Borrow (*) 228
16.7. Operations With Mixed 64-Bit and 32-Bit Operands 229
16.8. Logical-Arithmetic Compare Instructions 232
16.9. Retrieving and Setting the Program Mask (*) 234
16.10. Summary 235
17. Binary Shifting 242
17.1. Unit Shifts 243
17.2. Single-Length Logical Shifts 245
17.2.1. Three-Operand Shift Instructions 247
17.3. Double-Length Logical Shifts 248
17.4. Arithmetic Shift Instructions 252
17.5. Rotating Shifts 257
17.6. Calculated Shift Amounts 257
17.7. Bit-Length Constants (*) 259
17.8. Summary 260
18. Binary Multiplication and Division 264
18.1. Overview of Multiplication Instructions 264
18.2. Arithmetic (Signed) Multiplication Instructions 265
18.2.1. Double-Length Arithmetic Products 265
18.2.2. Single-Length Arithmetic Products 267
18.3. Logical (Unsigned) Multiplication Instructions 270
18.4. How Multiplication Is Done (*) 272
18.5. Division Instructions 274
18.6. Arithmetic (Signed) Division Instructions 275
18.6.1. Double-Length Division 275
18.6.2. Single-Length Division 278
18.7. Logical (Unsigned) Division Instructions 279
18.8. How Division Is Done (*) 280
18.9. Summary 283
19. Logical Operations 288
19.1. Logical Operations 289
19.2. Register-Based Logical Instructions 289
19.3. Logical AND 290
19.4. Logical OR 291
19.5. Logical Exclusive OR 292
19.6. Interesting Uses of Logical Instructions (*) 295
19.7. Summary 297

Chapter VI: Addressing, Immediate Operands, and Loops 301
20. Address Generation and Addressing Modes 302
20.1. Address Generation 302
20.1.1. Address Generation With 12-Bit Displacements 302
20.1.2. Address Generation With 20-Bit Displacements 302
20.1.3. Address Generation With Relative-Immediate Operands 305
20.2. Addressing Modes 307
20.3. Load Address Instructions 309
20.4. 64-Bit Virtual Addresses 314
20.5. Summary 314
21. Immediate Operands 316
21.1. Insert and Load Instructions with Immediate Operands 318
21.1.1. Logical-Immediate Insert Instructions 318
21.1.2. Arithmetic- and Logical-Immediate Load Instructions 318
21.2. Arithmetic Instructions with Immediate Operands 321

21.2.1. Arithmetic-Immediate Add and Subtract Instructions 321
21.2.2. Arithmetic-Immediate Compare Instructions 322
21.2.3. Arithmetic-Immediate Multiply Instructions 322
21.3. Logical Operations with Immediate Operands 323
21.3.1. Logical-Immediate AND Instructions 323
21.3.2. Logical-Immediate OR Instructions 323
21.3.3. Logical-Immediate XOR Instructions 324
21.4. Summary 325
22. Branches, Loops, and Indexing 329
22.1. Branch Relative on Condition Instructions 329
22.2. A Simple Example of a Loop 331
22.3. Simple Tables and Array Indexing 332
22.4. Branch on Count Instructions 334
22.5. Looping in General 338
22.6. Branch on Index Instructions 340
22.7. Examples Using BXLE 343
22.8. Examples Using BXH 346
22.9. Specialized Uses of BXH and BXLE (*) 347
22.10. Summary 349

Chapter VII: Bit and Character Data **351**
23. Bit and Byte Data and Instructions 352
23.1. SI- and SIY-Type Instructions 352
23.2. MVI Instructions 353
23.3. NI, OI, and XI Instructions 353
23.4. CLI Instructions 354
23.5. Test Under Mask Instructions 356
23.6. Bit Data 358
23.7. Avoiding Bit-Naming Problems (*) 359
23.8. A Data Conversion Example 361
23.9. Instruction Modification (*) 361
23.10. Summary 363
24. Character Data and Basic Instructions 365
24.1. Basic SS-Type Instructions 365
24.2. Operand Specifications and Explicit Lengths 366
24.3. Symbol Length Attribute References 368
24.4. Implied Lengths 368
24.5. The Encoded Length “L” and Program Length “N” 370
24.6. The MVC and MVCIN Instructions 372
24.6.1. MVC: Move Characters 372
24.6.2. MVCIN: Move Characters Inverse 373
24.6.3. MVCOS: Move Characters With Optional Specifications (*) 374
24.7. The NC, OC, and XC Instructions 376
24.8. The CLC Instruction 378
24.9. The TR (translate) Instruction 379
24.10. The TRT and TRTR Instructions 383
24.10.1. TRT 384
24.10.2. TRTR 387
24.11. The Execute Instructions 389
24.11.1. Execute Instruction Without Target-Instruction Modification 390
24.11.2. Execute Instruction with Target-Instruction Modification 391
24.11.3. Comments on the Execute Instructions (*) 392
24.11.4. Modifiable Parts of Instructions 393
24.12. Summary 396
25. Character Data and Extended Instructions 403
25.1. Move Long and Compare Logical Long 403
25.1.1. MVCL 405
25.1.2. CLCL 407
25.2. Move Long and Compare Logical Long Extended 410
25.2.1. MVCLE 411
25.2.2. CLCLE 413
25.3. Special “C-String” Instructions 415

25.4. Search String Instruction . . . 415
25.5. Move String Instruction . . . 417
25.6. Compare Logical String Instruction . . . 419
25.7. Translate Extended Instruction . . . 421
25.8. Compare Until Substring Equal Instruction (*) . . . 423
25.9. Summary . . . 425
26. Other Types of Character Data (*) . . . 428
26.1. Character Representations . . . 428
26.1.0. An Early Character Encoding . . . 428
26.1.1. BCD characters . . . 429
26.2. EBCDIC Representations and Code Pages . . . 430
26.3. ASCII . . . 432
26.4. Double-Byte EBCDIC Data (*) . . . 434
26.4.1. The DBCS Option (*) . . . 436
26.4.2. G-Type DBCS Constants and Self-Defining Terms (*) . . . 436
26.4.3. Continuation Rules for DBCS Data (*) . . . 437
26.5. Unicode . . . 438
26.5.1. The Unicode Representation . . . 438
26.5.2. Glyphs and Characters . . . 439
26.5.3. Unicode Character Constants . . . 439
26.6. Unicode Instructions . . . 441
26.6.1. String Search, Move, and Compare . . . 441
26.6.2. Optional Operands (*) . . . 443
26.6.3. Translation . . . 444
26.6.4. Conversion Among Transformation Formats (*) . . . 447
26.7. Translate and Test Extended . . . 450
26.8. Byte Reversal and Workstation Data . . . 453
26.8.1. Byte-Reversing Instructions . . . 453
26.9. Summary . . . 456

Chapter VIII: Zoned and Packed Decimal Data and Operations . . . 459
27. Zoned and Packed Decimal Representations . . . 460
27.1. Zoned Decimal Representation . . . 460
27.1.1. Why Zoned Decimal Is The Way It Is (*) . . . 463
27.2. Zoned Decimal Constants . . . 464
27.3. Packed Decimal Representation . . . 465
27.4. Packed Decimal Constants . . . 467
27.4.1. Scale Attributes and Packed Decimal Constants (*) . . . 467
27.5. Converting Between Packed and Zoned . . . 469
27.6. The PACK Instruction . . . 471
27.7. The UNPK Instruction . . . 474
27.8. Packing and Unpacking ASCII and Unicode Data (*) . . . 478
27.8.1. Packing ASCII and Unicode Data . . . 478
27.8.2. Unpacking ASCII and Unicode Data . . . 479
27.9. Printing Hexadecimal Values . . . 481
27.10. Summary . . . 483
28. Packed Decimal Arithmetic . . . 484
28.1. General Rules . . . 484
28.1.1. Precision and Accuracy . . . 485
28.2. Decimal Addition and Subtraction . . . 485
28.3. Decimal Comparison . . . 487
28.4. Decimal Multiplication . . . 489
28.5. Decimal Division . . . 490
28.6. True Decimal Addition (*) . . . 492
28.7. Complement Decimal Addition (*) . . . 493
29. Packed Decimal Instructions . . . 497
29.1. TP Instruction . . . 498
29.2. ZAP Instruction . . . 499
29.3. AP and SP Instructions . . . 501
29.4. CP Instruction . . . 503
29.5. MP Instruction . . . 506
29.6. DP Instruction . . . 509

29.7. SRP Instruction . . . 511
29.7.1. Biased and Unbiased Rounding with SRP (*) . . . 513
29.8. MVO Instruction . . . 516
29.9. Decimal Shifting Using MVO (*) . . . 518
29.9.1. Shift Right an Odd Number of Digits . . . 518
29.9.2. Shift Left an Odd Number of Digits . . . 519
29.9.3. Shifting an Even Number of Digits . . . 519
29.9.4. Shifting Left an Even Number of Digits . . . 520
29.9.5. Shifting Right an Even Number of Digits . . . 520
29.10. Scaled Packed Decimal Computations: General Rules . . . 522
29.10.1. Precision and Scale . . . 522
29.10.2. General Rules: Addition and Subtraction . . . 523
29.10.3. General Rules: Multiplication . . . 523
29.10.4. General Rules: Division (*) . . . 524
29.10.5. COBOL and PL/I Notations (*) . . . 525
29.11. Example of a Packed Decimal “Business” Computation . . . 526
29.11.1. The Wholesaler's Calculation . . . 526
29.11.2. The Retailer's Calculation . . . 527
29.11.3. Comments . . . 529
29.11.4. Using Integer and Scale Attributes (*) . . . 529
29.12. Summary . . . 530
30. Converting and Formatting Packed Decimal Data . . . 532
30.1. CVD, CVDY, and CVDG Instructions . . . 532
30.2. CVB, CVBY, and CVBG Instructions . . . 534
30.3. Editing Overview . . . 536
30.4. Simple Examples of Editing . . . 538
30.5. Single-Field Editing . . . 541
30.5.1. Editing Negative Values . . . 541
30.5.2. Protecting High-Order Fields . . . 542
30.6. The EDMK Instruction . . . 543
30.7. Editing Multiple Fields (*) . . . 545
30.8. Summary Comments on Editing (*) . . . 546

Chapter IX: Floating-Point Data and Operations . . . 551
31. Floating-Point Numbers: Introduction . . . 552
31.1. Scaled Fixed-Point Arithmetic . . . 552
31.2. Mixed Integer-Fraction Representation . . . 553
31.2.1. Scaled Fixed-Point Binary Arithmetic (*) . . . 554
31.2.2. Scaled Fixed-Point Binary Constants (*) . . . 555
31.3. Converting Fractions Between Bases (*) . . . 557
31.4. Why Use Floating-Point Numbers? . . . 559
31.4.1. Precision and Accuracy . . . 560
31.5. Floating-Point Representations . . . 560
31.5.1. Left Normalization . . . 561
31.5.2. Right Normalization . . . 562
31.5.3. No Normalization . . . 562
31.5.4. Some Additional Details (*) . . . 562
31.6. System z Floating-Point Representations . . . 564
31.7. System z Floating-Point Registers . . . 564
31.8. Floating-Point Constants . . . 567
31.9. Representation-Independent Floating-Point Instructions . . . 568
31.9.1. Register-Storage Instructions . . . 568
31.9.2. Register-Register Instructions . . . 568
31.9.3. Load-Zero Instructions . . . 569
31.9.4. GPR-FPR Copying Instructions . . . 569
31.9.5. Sign-Copying Instruction . . . 570
31.10. Summary . . . 570
32. Basic Concepts of Floating-Point Arithmetic . . . 573
32.1. Floating-Point Multiplication . . . 573
32.2. Pre-Normalization of Fraction Operands . . . 574
32.3. Floating-Point Rounding . . . 574
32.4. Guard and Rounding Digits (*) . . . 575

32.5. Integer-Based Representations (*) . . . 577
32.6. Floating-Point Division . . . 578
32.7. Floating-Point Addition and Subtraction . . . 578
32.8. Floating-Point Precision . . . 580
32.9. Floating-Point Range . . . 582
32.10. Exponents and Characteristics . . . 584
32.11. Summary . . . 585
33. Hexadecimal Floating-Point Data and Operations . . . 586
33.1. Hexadecimal Floating-Point Data . . . 586
33.2. Writing Hexadecimal Floating-Point Constants . . . 590
33.2.1. Decimal Exponents . . . 591
33.3. Modifiers . . . 592
33.3.1. Length Modifiers . . . 592
33.3.2. Scale Modifiers (*) . . . 592
33.3.3. Exponent Modifiers . . . 593
33.4. Subtypes Q and H (*) . . . 594
33.4.1. LQ-Type Constants . . . 594
33.4.2. Subtype H . . . 595
33.4.3. Difficult Numbers (*) . . . 596
33.5. Basic Hexadecimal Floating-Point Instructions . . . 597
33.6. Hexadecimal Floating-Point RR-Type Data-Movement Instructions . . . 597
33.7. Hexadecimal Floating-Point Multiplication . . . 599
33.7.1. Exponent Overflow and Underflow. . . . 602
33.8. Hexadecimal Floating-Point Division . . . 603
33.8.1. The Halve Instructions (*) . . . 604
33.9. Hexadecimal Floating-Point Addition and Subtraction . . . 606
33.9.1. Unnormalized Addition and Subtraction . . . 609
33.9.2. Older Uses of Unnormalized Addition (*) . . . 609
33.10. Adding Operands of Like Sign (*) . . . 612
33.11. Adding Operands of Unlike Sign (*) . . . 612
33.11.1. Hexadecimal Floating-Point Complement Addition (*) . . . 613
33.11.2. Implementing Hexadecimal Floating-Point Complement Addition (*) . 614
33.12. Hexadecimal Floating-Point Comparison . . . 615
33.13. Rounding and Lengthening Instructions . . . 616
33.13.1. Rounding Instructions . . . 616
33.13.2. Lengthening Instructions . . . 618
33.14. Converting Between Binary Integers and HFP . . . 620
33.14.1. Converting Binary Integers to Hexadecimal Floating-Point . . . 620
33.14.2. Converting Hexadecimal Floating-Point to Binary Integers . . . 621
33.15. Hexadecimal Floating-Point Integers and Remainders (*) . . . 625
33.16. Square Root Instructions (*) . . . 626
33.17. Multiply and Add/Subtract Instructions (*) . . . 627
33.18. Some Hexadecimal Floating-Point History (*) . . . 629
33.18.1. Zeroing Floating-Point Registers . . . 629
33.18.2. Hexadecimal Floating-Point to Binary Conversion Comments (*) . . . 629
33.18.3. Initial System/360 Oversights . . . 630
33.19. Summary . . . 630
34. Binary Floating-Point Data and Operations . . . 638
34.1. Binary Floating-Point Data . . . 638
34.1.1. Data Representations . . . 639
34.1.2. Normal Numbers . . . 640
34.1.3. Special Values . . . 640
34.1.4. Range of the Representation . . . 641
34.2. Writing Binary Floating-Point Constants . . . 642
34.2.1. Decimal Exponents and Exponent Modifiers . . . 644
34.2.2. Length Modifiers (*) . . . 645
34.3. Binary Floating-Point Arithmetic in General . . . 646
34.3.1. Rounding Modes . . . 646
34.3.2. Denormalized Numbers . . . 647
34.3.3. Arithmetic with Zero, Infinity, and NaNs . . . 648
34.4. Binary Floating-Point Exceptions, Interruptions, and Controls . . . 649
34.4.1. Binary Floating-Point Exceptions (*) . . . 649

34.4.2. FPC Register Instructions (*) 651
34.4.3. Exception Actions (*) 651
34.4.4. Scaled Exponents (*) 653
34.5. Basic Binary Floating-Point Instructions 653
34.6. Binary Floating-Point RR-Type Data Movement Instructions 655
34.7. Binary Floating-Point Multiplication 657
34.8. Binary Floating-Point Division 659
34.9. Binary Floating-Point Addition and Subtraction 661
34.10. Binary Floating-Point Comparison 662
34.10.1. Compare and Signal (*) 663
34.11. Binary Floating-Point Rounding and Lengthening Instructions (*) 664
34.11.1. Rounding Instructions (*) 664
34.11.2. Lengthening Instructions (*) 664
34.12. Converting Between BFP and Binary Integers (*) 666
34.12.1. Converting Binary Integers to Binary Floating-Point (*) 666
34.12.2. Converting Binary Floating-Point to Binary Integers (*) 666
34.13. Binary Floating-Point Integers and Remainders (*) 668
34.13.1. Load FP Integer Instructions 668
34.13.2. Divide to Integer Instructions (*) 669
34.14. Binary Floating-Point Square Root Instructions (*) 671
34.15. Binary Floating-Point Multiply and Add/Subtract (*) 672
34.16. Summary 673
35. Decimal Floating-Point Data and Operations 680
35.1. Representations 681
35.1.1. Conceptual View of the Decimal Floating-Point Representation 682
35.2. System z Decimal Floating-Point Data Encoding and Representation (*) . . 684
35.2.1. Decimal Floating-Point Data Encoding (*) 685
35.2.2. Decimal Floating-Point Data Representation (*) 686
35.2.3. Decimal Floating-Point Combination Field (*) 687
35.3. Decimal Floating-Point Constants 690
35.3.1. Rounding-Mode Suffixes for Decimal Floating-Point Constants 691
35.3.2. Decimal Exponents and Modifiers 692
35.4. Decimal Floating-Point Data Classes (*) 693
35.5. Decimal Floating-Point Operations: Rounding, Quanta, and Exceptions . . . 695
35.5.1. Rounding 695
35.5.2. Preferred Exponent and Quantum 696
35.5.3. DFP Exceptions 698
35.4.4. Overflow/Underflow Scale Factors (*) 699
35.6. Decimal Floating-Point Data Movement Instructions 699
35.6.1. Copy Sign 699
35.6.2. Copy between General and Floating-Point Registers 700
35.6.3. Copy Among Floating-Point Registers 700
35.7. Decimal Floating-Point Arithmetic Instructions 701
35.7.1. Multiplication 702
35.7.2. Division 703
35.7.3. Addition and Subtraction 703
35.8. Decimal Floating-Point Compare Instructions 705
35.8.1. Compare 705
35.8.2. Compare and Signal 705
35.8.3. Compare Biased Exponent 706
35.9. Converting Decimal Floating-Point To and From Fixed Binary 707
35.9.1. Convert From Fixed Binary To DFP 707
35.9.2. Convert From DFP To Fixed Binary 707
35.10. Converting Decimal Floating-Point To/From Packed and Zoned Decimal . 709
35.10.1. Convert To/From Signed Packed Decimal 709
35.10.2. Convert To/From Unsigned Packed Decimal 711
35.10.3. Convert To/From Zoned Decimal 712
35.11. Decimal Floating-Point Load Operations 714
35.11.1. Load and Test, Complement, Negative, and Positive 714
35.11.2. Load Floating-Point Integer 715
35.11.3. Load Lengthened 716
35.11.4. Load Rounded 717

35.12. Decimal Floating-Point Miscellaneous Operations (*) 718
35.12.1. Set Decimal Rounding Mode 718
35.12.2. Extract and Insert Biased Exponent 719
35.12.3. Extract Significance 720
35.12.4. Shift Significand Left/Right 720
35.12.5. Quantize 722
35.12.6. Reround 724
35.12.7. Decimal Floating-Point Data Groups (*) 726
35.13. Example of a Decimal Floating-Point “Business” Computation 728
35.13.1. The Wholesaler's Calculation 728
35.13.2. The Retailer's Calculation 729
35.13.3. Comparing Packed and Floating Decimal 729
35.14. Decimal Floating-Point Binary-Significand Format (*) 730
35.15. Summary 731
36. Floating-Point Summary 739
36.1. Floating-Point Data Representations 739
36.2. Floating-Point Properties 741
36.3. Floating-Point Exceptions 741
36.4. Defining Floating-Point Constants 742
36.5. Converting Among Decimal, Hexadecimal and Binary Representations . . . 743
36.5.1. In-Out Conversions 743
36.5.2. Out-In Conversions 744
36.5.3. The PFPO Instruction (*) 745
36.6. “Real” and “Realistic” (Floating-Point) Arithmetic 745
36.7. When Does Zero Not Behave Like Zero? (*) 747
36.7.1. Hexadecimal Floating-Point 748
36.7.2. Binary Floating-Point 748
36.7.3. Decimal Floating-Point 749
36.8. Examples of Former Floating-Point Representations and Behaviors (*) . . . 749
36.9. Summary 751

Chapter X: Large Programs and Modularization **755**
37. Subroutines and Linkage Conventions 756
37.1. Basic Concepts 756
37.1.1. Linkage 757
37.1.2. The Branch and Save Instructions 757
37.1.3. Argument Passing 759
37.1.4. Returned Values 762
37.1.5. Status Preservation 763
37.2. A General Linkage Convention 765
37.3. Argument Passing 766
37.3.1. Variable-Length Argument Lists 767
37.3.2. Argument Lists with 64-Bit Addresses 768
37.4. Save Areas 770
37.4.1. Extended Save Area Conventions (*) 773
37.4.2. Format-4 Save Area Conventions for 64-bit Registers (*) 773
37.4.3. Format-5 Save Area Conventions for 32- and 64-bit Registers (*) 774
37.5. Additional Conventions (*) 777
37.5.1. Entry Point Identifiers (*) 777
37.5.2. Calling Point Identifiers (*) 778
37.5.3. Save Area Return Flags (*) 778
37.5.4. Return Codes (*) 779
37.5.5. Conventions for Floating-Point Registers 782
37.5.6. Main-Program Parameters 782
37.6. Assisted Linkage (*) 783
37.7. Lowest Level Subroutines 785
37.8. Summary 787
37.8.1. Standard Linkage Conventions 787
38. Large Programs, Control Sections, and Linking 790
38.1. Uniform Addressability for Large Programs 790
38.1.1. Other Techniques (*) 792
38.2. Simplifying Addressability Problems in Large Programs 797

38.2.1. Internal Subroutines Without Local Addressability 797
38.2.2. Internal Subroutines With Local Addressability 798
38.2.3. Minimizing the Number of Base Registers 799
38.2.4. Relative Branches, Immediate Operands, and Long Displacements . . . 800
38.2.5. Separating Instructions and Data 800
38.3. Separate Assemblies 802
38.4. Control Sections 803
38.4.1. Resuming Control Sections 806
38.4.2. Literals in Multi-Section Assemblies (*) 808
38.4.3. Location Counter Discontinuities (*) 808
38.4.4. Section Alignment (*) 809
38.4.5. Threaded Location Counters (*) 809
38.4.6. The "Location Counter" Instruction LOCTR (*) 810
38.5. External Symbols 818
38.5.1. EXTRN and WXTRN Statements 819
38.5.2. V-Type Address Constants 820
38.5.3 ENTRY Statement 821
38.5.4. The External Symbol Dictionary Listing 824
38.5.5. External Symbol Addressing and Residence Modes 827
38.6. Object Modules 831
38.6.1. Relocation Dictionary and External Symbol Dictionary 832
38.7. Program Linking: Combining Object Modules 833
38.7.1. Assigning COMMON Sections 836
38.7.2. Relocating Address Constants 836
38.7.3. External Dummy Sections (*) 838
38.7.4. Loading Object Modules (*) 841
38.8. Load Modules and Program Objects 845
38.8.1. External Subroutines and Assisted Linkage: Overlay (*) 847
38.8.2. Program Objects (*) 848
38.8.3. The "Class Attribute" Instruction CATTR 851
38.8.4. Programming for Program Objects 854
38.8.5. Comparing Load Modules and Program Objects 854
38.9. Loading Saved Modules into Storage 855
38.9.1. Loading Load Modules 855
38.9.2. Loading Program Objects 856
38.10. Changing Addressing Modes 858
38.10.1. The BASSM Instruction 859
38.10.2. The BSM Instruction 860
38.10.3. Branch and Return With Addressing Mode Change 861
38.10.4. Load Logical Thirty-One Bits Instructions 863
38.11. Summary 866

Chapter XI: Dummy Sections, Enhanced USINGs, and Data Structures 871
39. Dummy Control Sections and Enhanced USING Statements 872
39.1. Dummy Control Sections 872
39.2. Multiple Data Structures 875
39.3. Shortcomings of Ordinary USING Statements 876
39.3.1. Ordinary USINGs 877
39.4. Labeled USING Statements and Qualified Symbols 882
39.4.1. Qualified Symbols 882
39.4.2. Dropping a Labeled USING Statement 883
39.4.3. Labeled USING Statement Summary 883
39.5. Dependent USING Statements 885
39.5.1. Definition of Dependent USING Statements 886
39.5.2. Examples of Dependent USING Statements 886
39.5.3. Mapping a CSECT as a DSECT 889
39.5.4. Dropping Dependent USINGs 890
39.5.5. Dependent USING Statement Summary 890
39.6. Labeled Dependent USING Statements 891
39.6.1. Nesting Structures Addressed with Ordinary USINGs 892
39.6.2. Nesting Structures Addressed with Labeled USINGs 892
39.6.3. Nested Structures Addressed with Labeled Dependent USINGs 892

39.6.4. Multiple Nesting of Identical Structures 893
39.6.5. Mapping an Array of Identical Data Structures 895
39.6.6. Two MVS Data Control Blocks (DCBs) in a Program 896
39.7. Example of a Large “Personnel-File” Record (*) 897
39.7.1. Personnel-File Record Example: Comparing Birth Dates 902
39.7.2. Personnel-File Record Example: Comparing Different Dates 902
39.7.3. Personnel-File Record Example: Copying Addresses 903
39.8. Summary 904
39.8.1. USING Statement Summary 904
39.8.2. DROP Statement Summary 905
40. Basic Data Structures 907
40.1. One-Dimensional Arrays 908
40.2. Two-Dimensional Arrays 910
40.3. General Array Subscripts 913
40.3.1. Multi-Dimensional Arrays (*) 913
40.3.2. Non-Homogeneous Arrays (Tables) 914
40.4. Address Tables 917
40.5. Searching an Ordered Array 919
40.6. Stacks 923
40.6.1. An Example Using a Stack 923
40.6.2. An Example Implementing a Stack 924
40.7. Lists 927
40.7.1. List Insertion 927
40.7.3. List Deletion 929
40.7.4. Free Storage Lists 929
40.8. Queues 934
40.9. Trees 937
40.10. Hash Tables 941
40.11. Summary 944

Chapter XII: System Services, Reenterability, and Recursion 949
41. Using System Services 950
41.1. Invoking System Services 950
41.2. Invoking System Services with Macro Instructions 951
41.3. Macro Formats: Standard, List, and Execute 952
41.3.1. List form with Empty Argument List 953
41.3.2. Register Forms and Arguments 954
41.3.3. MODE=24, MODE=31 954
41.3.4. Mixed Case Macro Arguments 955
41.3.5. The SYSSTATE Macro 955
41.4. Causing Abnormal Termination 956
41.5. Storage Management 957
41.5.1. The GETMAIN Macro 958
41.5.2. The FREEMAIN Macro 959
41.5.3. The STORAGE Macro 960
41.5.4. Subpools (*) 961
41.5.5. Optional Operands (*) 961
41.6. Basic Input and Output 962
41.6.1. A Simple Scenario 962
41.6.2. Access Techniques and Access Methods 966
41.6.3. The Data Control Block (DCB) 966
41.6.4. Important Record Formats 968
41.6.5. Opening the DCB 969
41.6.6. Closing the DCB 970
41.6.7. The DCBD Macro and the IHADCB Dummy Section 970
41.6.8. The DCBE Macro and 31-bit Address Mode 971
41.6.9. I/O Summary 971
41.6.10. A Sample Program 971
41.7. Handling Program Interruptions 972
41.7.1. Program Interruptions 973
41.7.2. Establishing a Program Interruption Exit 973
41.7.3. Terminating a Program Interruption Exit 974

41.7.4. Handling a Program Interruption . . . 975
41.8. Abnormal Terminations of Any Kind . . . 976
41.8.1. The ESTAE Macro . . . 978
41.8.2. Interruption Processing . . . 979
41.8.3. Percolation and Retry . . . 979
41.8.4. Summary . . . 980
41.9. Summary . . . 981
42. Reenterability and Recursion . . . 983
42.1. Reenterability . . . 983
42.1.1. What it Means in General . . . 983
42.1.2. What it Means in Practice . . . 984
42.1.3. Assembly-Time Considerations . . . 984
42.1.4. At Linking Time . . . 984
42.1.5. Techniques . . . 985
42.2. Recursion . . . 987
42.3. Summary . . . 992

Appendix A: Conversion and Reference Tables . . . 995
Hexadecimal Digits in Decimal and Binary . . . 995
Hexadecimal Addition and Multiplication Tables . . . 996
Powers of 2 . . . 997
Multiples of Powers of Sixteen . . . 1000
Powers of 10 in Hexadecimal . . . 1001
Hexadecimal and Decimal Integers . . . 1003
Conversion Tables for Hexadecimal Fractions . . . 1011
EBCDIC Character Representation in Assembler Language Programs . . . 1012
ASCII Character Representation in Assembler Language Programs . . . 1013
DC Statement Types . . . 1014

Appendix B: Simple I/O Macros . . . 1015
B.1. Macro Facilities . . . 1015
B.1.1. The CONVERTI Macro Instruction . . . 1016
B.1.2. The CONVERTO Macro Instruction . . . 1017
B.1.3. The DUMPOUT Macro Instruction . . . 1018
B.1.4. The PRINTLIN Macro Instruction . . . 1018
B.1.5. The PRINTOUT Macro Instruction . . . 1019
B.1.6. The READCARD Macro Instruction . . . 1020
B.1.7. PRINTOUT and DUMPOUT Header . . . 1020
B.1.8. Usage Notes . . . 1021
B.2. Sample Program . . . 1021
B.3. The Macro Instruction Definitions . . . 1022
B.3.1. Operating System Environment and Installation Considerations . . . 1023
B.4.1. CONVERTI Macro Definition . . . 1024
B.4.2. CONVERTO Macro Definition . . . 1024
B.4.3. DUMPOUT Macro Definition . . . 1025
B.4.4. PRINTLIN Macro Definition . . . 1026
B.4.5. PRINTOUT Macro Definition . . . 1026
B.4.6. READCARD Macro Definition . . . 1028
B.4.7. $$GENIO Macro Definition . . . 1028

Glossary of Terms and Abbreviations . . . 1041

Bibliography . . . 1057
Basic References . . . 1057
System/360 Architecture History . . . 1058
Assembler Design and Implementation . . . 1058
Other General References . . . 1058

Acknowledgments . . . 1059

Notices . . . 1061

Trademarks . . . 1061

Suggested Solutions to Selected Exercises and Programming Problems . . . 1063
Section 1 Solutions . . . 1064
Section 2 Solutions . . . 1065
Section 3 Solutions . . . 1071
Section 4 Solutions . . . 1072
Section 5 Solutions . . . 1074
Section 6 Solutions . . . 1076
Section 7 Solutions . . . 1077
Section 8 Solutions . . . 1079
Section 9 Solutions . . . 1082
Section 10 Solutions . . . 1083
Section 11 Solutions . . . 1085
Section 12 Solutions . . . 1086
Section 13 Solutions . . . 1088
Section 14 Solutions . . . 1093
Section 15 Solutions . . . 1096
Section 16 Solutions . . . 1098
Section 17 Solutions . . . 1112
Section 18 Solutions . . . 1124
Section 19 Solutions . . . 1137
Section 20 Solutions . . . 1142
Section 21 Solutions . . . 1145
Section 22 Solutions . . . 1148
Section 23 Solutions . . . 1156
Section 24 Solutions . . . 1158
Section 25 Solutions . . . 1170
Section 26 Solutions . . . 1175
Section 27 Solutions . . . 1180
Section 28 Solutions . . . 1187
Section 29 Solutions . . . 1189
Section 30 Solutions . . . 1197
Section 31 Solutions . . . 1204
Section 32 Solutions . . . 1209
Section 33 Solutions . . . 1212
Section 34 Solutions . . . 1223
Section 35 Solutions . . . 1230
Section 36 Solutions . . . 1238
Section 37 Solutions . . . 1241
Section 38 Solutions . . . 1249
Section 39 Solutions . . . 1261
Section 40 Solutions . . . 1264
Section 41 Solutions . . . 1275
Section 42 Solutions . . . 1278

Index . . . 1281

Figures

1. Example of numbering and notation . . . 12
2. One stage of a binary adder . . . 35
3. “Circular” representation of two's complement representation . . . 36
4. Conceptual structure of a typical computer . . . 42
5. Conceptual structure of System z . . . 43
6. A byte containing 8 binary digits . . . 43
7. A portion of memory, with addresses shown above each byte . . . 43
8. A portion of memory . . . 44

9. A single 64-bit general register . . . 45
10. All sixteen general registers . . . 46
11. Four Floating-Point Registers . . . 47
12. Sketch of a Program Status Word . . . 47
13. Basic instruction cycle . . . 50
14. Instruction formats and data interactions . . . 52
15. Opcode bit patterns for typical instruction types . . . 53
16. Instruction cycle with interruptions . . . 55
17. Typical instruction format for old computers . . . 61
18. Structure of an addressing halfword . . . 62
19. Sketch of Effective Address calculation . . . 62
20. RX-type instruction, showing index register specification digit . . . 64
21. Sketch of Effective Address calculation with indexing . . . 64
22. 31-bit Virtual Address . . . 68
23. Simple view of Assembler processing . . . 72
24. Simple view of program linking . . . 73
25. Simple view of program loading and execution . . . 74
26. Assembler Language statement columns . . . 75
27. Comment statement examples . . . 76
28. Block comments . . . 76
29. Statement fields for machine, assembler, and macro-instruction statements . . . 77
30. A machine instruction statement . . . 78
31. An assembler instruction statement . . . 78
32. The macro-instruction statement RETURN . . . 78
33. A complete Assembler Language program . . . 81
34. RX Instruction with explicit operands . . . 109
35. A simple program segment . . . 117
36. Simple program segment with assembled contents . . . 118
37. Same program segment, at different memory addresses . . . 118
38. Same program segment, with assembled contents . . . 118
39. Program segment with pre-calculated explicit base and displacements . . . 119
40. Program segment with explicit base and Assembler-calculated displacements . . . 119
41. Program Segment with USING Instruction . . . 120
42. Sample program segment with erroneous statement . . . 122
43. Sketch of pass one of an assembly . . . 124
44. Sketch of pass two of an assembly . . . 125
45. USING Table with one entry . . . 126
46. Program segment with second USING statement . . . 127
47. USING Table with multiple entries . . . 127
48. Assembled contents when two USINGs are active . . . 128
49. USING Table after DROP statement . . . 129
50. USING Table after second DROP statement . . . 130
51. Implied and explicit length specifications . . . 140
52. Multiple constants . . . 142
53. F-type constant with decimal exponent . . . 146
54. Character, hexadecimal, and binary constants . . . 150
55. Length attribute reference to two constants, one a literal . . . 155
56. Describing fields of a (U.S.) telephone number . . . 160
57. Describing fields of an Assembler Language statement . . . 161
58. Define a group of words . . . 161
59. Describing fields of an Assembler Language statement using ORG instructions . . . 167
60. Describing an Assembler symbol cross-reference listing line . . . 171
61. 32-bit portion of a 64-bit general register . . . 178
62. Sign extension by LH instruction . . . 183
63. Loss of significant digits using STH/LH . . . 183
64. Loss of significant digits using STH/LH . . . 183
65. Action of IC and STC instructions . . . 185
66. Interchanging two bytes with IC and STC . . . 185
67. Inserting a small number into a register . . . 185
68. Examples of some RR-type instructions . . . 188
69. 64-bit general register . . . 189
70. Sign extension by LGH instruction . . . 191

71. Examples of some RR-type instructions for 64-bit operands 192
72. Sign extension by LHR instruction 193
73. Sign extension for instructions with mixed 32- and 64-bit signed operands 195
74. Sign extension by Load Byte instructions 196
75. Zero extension by Load Logical Character instructions 197
76. Operation of Load Logical Halfword instructions 197
77. Operation of Load Logical word instructions 197
78. Operation of Load Logical Thirty One Bits instructions 198
79. Examples of conditional branch instructions 205
80. CNOP alignments and operands 209
81. Calculate a sum with an intermediate test 217
82. Calculate the sum of the first N odd integers 218
83. Example of arithmetic addition and subtraction 218
84. Testing the result of arithmetic instructions 218
85. Calculate a 64-bit sum with an intermediate test 221
86. Adding two 64-bit numbers 221
87. Examples of arithmetic comparisons 222
88. Calculate the sum of N odd integers 223
89. Adding two 64-bit numbers logically 225
90. Double-length complementation 225
91. Double-length complementation, a simpler way 226
92. Double-length addition 226
93. Double-length subtraction 226
94. Example of logical addition and subtraction 227
95. Double-length addition with carry 229
96. Double-length subtraction with borrow 229
97. Sign extension for instructions with mixed 32- and 64-bit signed operands 230
98. Calculate a 64-bit sum with an intermediate test 230
99. Calculate a 64-bit sum with an intermediate test 230
100. Sign extension for instructions with mixed 32- and 64-bit unsigned operands 231
101. Examples of logical comparisons 232
102. Comparing logically ordered values 233
103. Bit positions used by IPM and SPM instructions (System/360 PSW sketch) 234
104. Register contents before shifting 243
105. Logical unit shift left 244
106. Logical unit shift right 244
107. Arithmetic unit shift right 244
108. Arithmetic unit shift left 244
109. Rounding an integer to the next higher multiple of 8 246
110. A 6-byte data entry 246
111. Storage definitions for a 6-byte data entry 246
112. Using shift instructions for a 6-byte data item 246
113. Shifting to make the low-order bit one (1) 248
114. Shifting to make the low-order bit one (2) 249
115. Four integers packed in a 32-bit word 249
116. Extracting one packed integer from a 32-bit word 249
117. Unpacking four unsigned integers using right shifts 250
118. Unpacking four unsigned integers using left shifts 250
119. Unpacking four signed integers 254
120. Logical rotate unit shift 257
121. Packing four unsigned bit-length constants in a 32-bit word 260
122. Packing four signed bit-length constants in a 32-bit word 260
123. General layout of multiplication operands 265
124. Double-length product of multiply operations 266
125. Calculate the sum of the first 10 cubed integers 267
126. Illustration of binary multiplication 273
127. General result of divide operation 274
128. Operands of double-length division 275
129. Example of division by 3 276
130. Example of rounded integer division 276
131. Example of rounded integer division with signed dividend 277
132. Ensuring a valid arithmetic division 277

133. Causing a fixed-point divide interruption 277
134. Operands of single-length division before division 278
135. Operands of single-length division after division 278
136. Example of logical division 280
137. Illustration of binary division 282
138. Logical operations AND, OR, and XOR 289
139. Examples of logical operations 290
140. Inserting a new integer value using AND and OR 291
141. Data masking using Exclusive OR 292
142. Rounding to the next multiple of 8 293
143. Rounding to the next multiple of 8 293
144. Complementing a double-length integer 293
145. Effective Address generation for long-displacement instructions 303
146. Addressability range with 12-bit displacements 304
147. Addressability range with 20-bit displacements 304
148. Effective Address formation for relative-immediate instructions 305
149. Areas of memory addressed by three AMODEs 308
150. System z PSW showing addressing-mode bits 308
151. Loading integer constants with the LAY instruction 310
152. Counting number of shifts to make rightmost bit a 1-bit 311
153. Using LA to set a branch address 311
154. 64-bit Virtual Address 314
155. 64-bit Virtual Address with Region Indexes 314
156. Instruction classes, including RI, RIL 316
157. Four halfwords in a 64-bit general register 317
158. Operation of six Insert Immediate instructions 318
159. Operation of LHI instruction 319
160. Operation of LGHI instruction 319
161. Examples of load-immediate instructions 320
162. Operation of six logical load instructions 320
163. Extracting an unsigned integer value using AND Immediate 323
164. Inserting a new integer value using AND Immediate 323
165. Data masking using immediate operands 324
166. Data masking using a symbolically defined immediate operand 324
167. A simple loop to scan and replace characters 332
168. A simple loop, using indexing 333
169. Indexing into a branch table 333
170. A backward loop to scan and replace characters 334
171. Calculate the sum of the first N odd integers 335
172. Store the cubes of the first 10 integers 336
173. Sketch of a Do-Until loop 339
174. Sketch of a Do-While loop 339
175. Store the cubes of the first 10 integers in a different way 340
176. Operation of BXH and BXLE instructions 342
177. Operation of BXH and BXLE instructions 342
178. Replacing special characters with blanks, using BXLE 343
179. Creating a table of cubed integers using BXLE 344
180. Creating a table of cubed integers using BXLE 344
181. Creating a table of cubed integers with addresses as controls 344
182. Creating a table of cubed integers using BXH 346
183. Creating a table of cubed integers, using BXH in a special way 346
184. Examples of the MVI instruction 353
185. Examples of the NI instruction 354
186. Examples of the OI instruction 354
187. Example of the XI instruction 354
188. A simpler loop to scan and replace characters 355
189. Setting an overflow-indication flag bit 357
190. Adding alternate list elements twice 357
191. Defining bit names safely 360
192. Using safely-defined bit names 360
193. Converting a binary integer to characters 361
194. Adding alternate list elements twice, with program modification 362

195. Adding alternate list elements twice, without program modification 363
196. Assembler Language syntax of basic SS-type instructions 366
197. Examples of SS-type instruction operands 367
198. SS-type instruction using a Length Attribute reference 369
199. Examples of Length Specification Bytes 372
200. Emulated operation of MVC instruction 372
201. Example of Move Inverse instruction 373
202. Emulated operation of MVCIN instruction 374
203. Example of MVCOS instruction 375
204. Inserting bits in a word using logical SS-type instructions 377
205. Emulating the TR instruction 380
206. TR instruction to change special characters to blanks 380
207. Translating hex digits to EBCDIC characters (1) 381
208. Translating hex digits to EBCDIC characters (2) 381
209. Searching for punctuation characters using CLI 385
210. Searching for punctuation characters using TRT 385
211. Using TRT to validate numeric characters 385
212. Using TRT to scan for embedded quotations 386
213. Using TRT to scan a string of names and build an occurrence list 387
214. Using TRTR to validate numeric characters 387
215. Scanning a string backward using CLI 388
216. Scanning a string backward using TRTR 388
217. Executing a list of instructions 390
218. Executing a list of instructions 390
219. Constructing an executed instruction 391
220. Moving a string of bytes of unknown length 392
221. Register use by CLCL and MVCL 404
222. Conceptual execution of the MVCL instruction 406
223. Using MVCL to set a field to blanks 406
224. Moving a message with padding and length checking 406
225. Conceptual execution of the CLCL instruction 408
226. Using CLCL to test for blanks 408
227. Comparing two records without padding 409
228. Register use by MVCLE and CLCLE 411
229. Conceptual execution of the MVCLE instruction 412
230. Using MVCLE to set a field to blanks 413
231. Using MVCLE to initialize an area to zero 413
232. Conceptual execution of the CLCLE instruction 413
233. Using CLCLE to test for all blanks 414
234. Registers bounding the SRST search string 416
235. Conceptual execution of the SRST instruction 416
236. Conceptual execution of the MVST instruction 418
237. Moving a null-terminated string 418
238. Using MVST to isolate comma-separated tokens 418
239. Conceptual execution of the CLST instruction 420
240. Translating characters to upper case with TRE 422
241. Examples using the CUSE instruction 424
242. Fragment of an Institute-machine punched paper tape 428
243. Mixed single- and double-byte EBCDIC characters 434
244. Examples of DBCS data 435
245. Extended continuation for DBCS data 437
246. CU-type constant generating Unicode characters 440
247. Using MVCLU to initialize an area to Unicode spaces 442
248. Using CLCLU to test for Unicode spaces 443
249. Assembler instruction statement for RRF-type instructions with an optional operand 444
250. Using TRTT to translate from DBCS to Unicode 446
251. Translating a long string with TR and MVC, and with TROO 446
252. Bits of a UTF-16 Unicode character 447
253. Bits of a UTF-16 Unicode surrogate pair 447
254. Bits of a UTF-32 Unicode character from a UTF-16 surrogate pair 448
255. Example of using TRTE 451
256. Big-Endian storage representation of X'87654321' 453

257. Little-Endian storage representation of X'87654321' 453
258. Byte reversal by LRV, LRVR, and STRV instructions 454
259. Byte reversal by LRVH and STRVH instructions 454
260. Four integers packed in a Big-Endian 32-bit word 455
261. The same four integers packed in a Little-Endian 32-bit word 455
262. Zone and numeric digits of a byte 460
263. Example of MVN and MVZ instructions 461
264. Zoned decimal sign conventions 462
265. A zoned decimal number 462
266. Zoned decimal constants with implied lengths 464
267. Zoned decimal constants with explicit lengths 464
268. Representation of a packed decimal number 466
269. Packed decimal constants with implied lengths 467
270. Packed decimal constants with explicit lengths 467
271. Format of typical two-length SS-type instructions 469
272. Examples of assembled PACK and UNPK instructions 470
273. Zoned and packed forms of +12345 471
274. PACK instruction operation 471
275. Converting from zoned to packed decimal using PACK 471
276. Examples of the PACK instruction 472
277. Digit swap using PACK 472
278. Operation of the UNPK instruction 475
279. Example of an UNPK instruction 475
280. Examples of UNPK instructions 475
281. Digit swap using UNPK 476
282. Packing ASCII characters 479
283. Packing Unicode characters 479
284. Unpacking to ASCII and Unicode characters 480
285. Unpacking hex digits (incorrectly) 481
286. Unpacking hex digits (correctly) 482
287. Converting hex data to printable characters 482
288. Operands for packed decimal division 490
289. Assembler Language syntax of the TP instruction 498
290. Examples of the ZAP instruction 500
291. Using ZAP to initialize a table of packed decimal operands 500
292. Initializing a table of decimal numbers using MVC 500
293. Examples of the AP and SP instructions 502
294. Adding a table of 50 packed decimal numbers 502
295. Adding positive and negative items separately 504
296. Finding the largest item in a table 505
297. Example of decimal multiplication 506
298. Using MP to square a table of decimal numbers 507
299. Using ZAP to set correct decimal multiplicand length 507
300. Using ZAP to set correct decimal multiplicand length 507
301. Generating 0− using MP 508
302. Decimal division using DP 509
303. Decimal division using Length Attribute References for operands 509
304. Computing the average of a table of decimal numbers 510
305. Assembler Language format of SRP machine instruction statement 511
306. Shifting a decimal operand left 3 places using SRP 512
307. Shifting a decimal operand right 2 places using SRP 512
308. Shifting a decimal operand right 1 place with rounding using SRP 513
309. Shifting a decimal operand with an EXecuted SRP 513
310. Operation of the MVO instruction 516
311. Two Examples of MVO results 516
312. Shifting a decimal operand right an odd number of digits 518
313. Shifting a decimal operand right an odd number of digits 518
314. Shifting a decimal operand left an odd number of digits 519
315. Shifting a decimal operand left by one digit 519
316. Shifting a decimal operand left by three or more digits 519
317. Shifting a decimal operand left an even number of digits 520
318. Shifting a decimal operand left an even number of digits 520

319. Shifting a decimal operand left an even number of digits 520
320. Shifting a decimal operand right an even number of digits 520
321. Shifting a decimal operand right an even number of digits 521
322. Ensuring decimal point alignment for packed decimal addition 523
323. A business calculation in packed decimal, part 1 527
324. A business calculation in packed decimal, part 2 527
325. A business calculation in packed decimal, part 3 528
326. A business calculation in packed decimal, part 4 528
327. Integer and Scale Attributes 529
328. Using Scale Attributes in a SRP instruction 529
329. Converting a 64-bit binary integer to packed decimal 533
330. Using CVD to format page numbers 533
331. Converting decimal characters to binary 535
332. Sketch of an editing operation 536
333. Representation of an editing pattern 537
334. Convert a packed decimal integer to characters using UNPK 538
335. Convert a packed decimal integer to characters using ED 538
336. Converting a 32-bit binary integer to characters 540
337. Editing a binary integer with separating commas 541
338. Editing a signed number 542
339. Using field protection with ED 542
340. Edited result with a floating currency symbol 543
341. Edited result with a properly placed floating currency symbol 544
342. Integer value with optional sign and separating commas 544
343. Editing two packed decimal numbers into a single field 545
344. Editing multiple values 545
345. Logical-operation description of the editing process 547
346. ED and EDMK operation 548
347. A data item containing an integer value 552
348. A data item containing integer and fraction parts 552
349. Values with radix point outside the digits 553
350. Calculating a tax amount in scaled fixed decimal arithmetic 553
351. Calculating a tax amount in scaled fixed binary arithmetic 554
352. Two binary constants scaled by 2**28 555
353. Defining a scaled binary constant 10**12 555
354. Multiplying two scaled binary numbers 556
355. Examples of data with widely ranging values 559
356. A typical floating-point representation 560
357. An example of a floating-point representation using 4 decimal digits 561
358. Another example of a floating-point representation using 4 decimal digits 561
359. A floating-point representation showing left normalized and unnormalized values . . 561
360. A floating-point representation showing right normalized and unnormalized values . 562
361. A floating-point representation showing values without normalization 562
362. Floating-point numbers with signed exponent 563
363. Examples of approximate floating-point representations 563
364. Three floating-point data lengths 564
365. Four floating-point registers 564
366. All sixteen floating-point registers, showing register pairings 566
367. Integer-based representation of 73 in FPI(10,4) 577
368. Illustrating floating-point division corrective right shift 578
369. Exponent range of representable and computable values 583
370. Hexadecimal floating-point number representations 587
371. Quadword aligned constants and data 594
372. Hexadecimal floating-point constants with rounding suffixes 595
373. Examples of hexadecimal floating-point instructions 598
374. Example of LTXR instruction 598
375. Examples of extended-precision hexadecimal RR instructions 598
376. Short hexadecimal floating-point multiplication 599
377. Floating-point registers used for hexadecimal floating-point multiplication 600
378. Calculating a table of short hexadecimal floating-point products 600
379. Calculating a table of long hexadecimal floating-point products 600
380. Floating-point registers used for hexadecimal floating-point multiplication 601

381. Example of hexadecimal floating-point divide instructions 604
382. Example of hexadecimal floating-point divide instructions 604
383. Example of a hexadecimal floating-point halve instruction 605
384. Hexadecimal halve instruction causing underflow 605
385. Example of hexadecimal floating-point addition 606
386. Evaluating a hexadecimal floating-point expression 607
387. Evaluating a hexadecimal floating-point inner product 608
388. Evaluating a polynomial with hexadecimal floating-point arithmetic 608
389. Evaluating a quadratic polynomial 608
390. Converting a binary integer to hexadecimal floating-point 609
391. Converting a hexadecimal floating-point number to a binary integer 610
392. Rounding a long hexadecimal floating-point number to short 617
393. Rounded inner product of long HFP numbers 617
394. Manually rounding long to short (1) 618
395. Manually rounding long to short (2) 618
396. Manually rounding long to short (3) 618
397. Converting a 32-bit integer to short hexadecimal floating-point 620
398. Converting a 64-bit integer to three hexadecimal floating-point values 620
399. Early conversion of integer to hexadecimal floating-point 621
400. Format of a machine instruction statement for converting HFP to binary 621
401. Calculating a HFP remainder 625
402. Evaluating a hexadecimal floating-point remainder 626
403. Examples of HFP square root instructions 627
404. Three binary floating-point data representations 639
405. Range of the binary floating-point representation 641
406. A view of the binary floating-point representation 642
407. Examples of short binary floating-point constants 642
408. Examples of long and extended binary floating-point constants 643
409. Rounding indicators for binary floating-point constants 643
410. Examples of parameterized binary floating-point NaNs 644
411. Binary floating-point constants with decimal exponents and modifiers 645
412. Values representable with gradual underflow 647
413. Floating-Point Control (FPC) register 649
414. Examples of binary floating-point data movement instructions 656
415. Example of binary floating-point multiply instructions 657
416. Examples of binary floating-point multiplication overflow and underflow 658
417. Examples of binary floating-point multiply instructions 658
418. Example of binary floating-point denormalized product 658
419. Example of binary floating-point extended-precision operands 658
420. Examples of binary floating-point division 659
421. Examples of binary floating-point division overflow and underflow 660
422. Examples of binary floating-point addition and subtraction 661
423. Examples of binary floating-point comparison 663
424. Examples of binary floating-point compare and signal instructions 663
425. Examples of binary floating-point rounding instructions 664
426. Examples of BFP load lengthened instructions 665
427. Examples of BFP load lengthened instructions with NaNs 665
428. Examples of binary integer to binary floating-point instructions 666
429. Examples of converting binary floating-point fractions to integers with rounding . . . 667
430. Examples of Convert to Fixed instructions 667
431. Examples of load FP integer instructions 669
432. Examples of divide to integer instructions 670
433. Example of iterative divide to integer 670
434. Iterative execution of a divide to integer instruction 671
435. Examples of binary floating-point square root instructions 672
436. Example of binary floating-point multiply and add instructions 673
437. Hexadecimal and binary floating-point representations 681
438. Conceptual decimal floating-point representation 682
439. Three decimal floating-point representations of the same value 683
440. Decimal floating-point data representation 686
441. System z decimal floating-point representations 687
442. DFP constants with exponent modifiers and decimal exponents 692

443. Examples of decimal floating-point Test Data Class instructions 694
444. Illustration of decimal floating-point rounding candidates 695
445. Illustration of decimal floating-point rounding candidates near zero 695
446. Floating-Point Control (FPC) register 698
447. Examples of converting decimal floating-point to fixed binary 708
448. Examples of converting decimal floating-point to binary integer 709
449. Converting signed packed decimal to decimal floating-point 710
450. Converting decimal floating-point to signed packed decimal 710
451. Converting decimal floating-point to signed packed decimal 710
452. Converting unsigned packed decimal to decimal floating-point 711
453. Converting decimal floating-point to unsigned packed decimal 711
454. Effect of the mask operand on Convert from Zoned results 713
455. Examples of converting decimal floating-point to zoned 714
456. DFP arithmetic with short operands 717
457. Floating-Point Control Register showing Decimal Rounding Mode bits 718
458. Example of extracting DFP biased exponent 719
459. Example of inserting a biased DFP exponent 720
460. Examples of DFP Extract Significance instructions 720
461. Converting an extended decimal floating-point value to packed decimal 722
462. Calculate price plus tax 724
463. Correctly rounding a cost to two decimal digits 724
464. Example of a reround instruction 725
465. Example of rerounding arbitrary amounts 725
466. Examples of assembled DFP constants using rounding for reround 726
467. Example of DFP binary-significand format 730
468. Sketch of short binary-significand format 730
469. BCD-to-DPD encodings 734
470. DPD-to-BCD translation 735
471. Degraded precision in adding hexadecimal floating-point pseudo-zeros 748
472. Trivial example of a subroutine (1) 757
473. Trivial example of a subroutine (2) 757
474. Subroutine linkage using a BAS instruction 758
475. Subroutine linkage using a BASR instruction 759
476. Subroutine linkage using an address constant 759
477. Simple shift subroutine (1) 760
478. Simple shift subroutine (2) 760
479. Simple shift subroutine with named arguments (3) 760
480. Simple shift subroutine (4) using argument addresses 761
481. Simple shift subroutine (5) with argument addresses in memory 761
482. Subroutine call with inline arguments 761
483. Subroutine returning past inline argument 762
484. Subroutine call with inline argument addresses 762
485. Subroutine with argument address list 763
486. Subroutine saves and restores registers 764
487. General argument-passing scheme 766
488. Subroutine call using an argument address list 766
489. Subroutine called with an argument address list 766
490. Constructing an argument address list 767
491. Two variable-length argument lists 767
492. Calling a subroutine with a variable-length argument list 767
493. Subroutine called with a variable-length argument list 767
494. Sketch of a variable-length argument list 768
495. Sample 64-bit argument list addresses 768
496. Standard save area layout 771
497. Sample subroutine calling sequence 771
498. Save area chaining instructions 772
499. Chained save areas 772
500. Reloading registers and returning to a caller 772
501. Format-4 save area layout 774
502. Example of using a Format-4 save area 774
503. Format-5 save area layout 775
504. Saving registers using a Format-5 save area 776

505. Return from a routine using a Format-5 save area 776
506. Example of an entry point identifier 777
507. Example of two calling point identifiers 778
508. Setting a return flag 778
509. Setting a return code in register 15 779
510. Testing a return code returned in register 15 780
511. Using a return code as a branch index 780
512. Using a return code as a branch index with relative branch instructions 780
513. Checking for valid return code values 780
514. Setting a reason code in register 0 781
515. Using RETURN macros to set return flags and return codes 781
516. Returning to an error branch without a return code 781
517. Call with error branch instructions 782
518. Convention for passing main-program parameters 782
519. Example of calling with assisted linkage 784
520. Example of a routine to implement assisted linkage 784
521. Assisted linkage routine with counters 784
522. Example of a lowest level subroutine 786
523. Establish three base registers (1) 791
524. Establish three base registers (2) 791
525. Establish three base registers (3) 791
526. Establish three base registers (4) 792
527. Establish three base registers (5) 792
528. Establish three base registers with risks (6) 792
529. Establish three base registers (7) 793
530. Establish three base registers (8) 793
531. Establish three base registers (9) 793
532. Calling a subroutine not needing local addressability 798
533. Calling a subroutine not locally addressable 799
534. Subroutine with local addressability 799
535. Replacing based branch instructions with relative-immediates 800
536. Replacing a based EXecute instruction with EXRL 800
537. Replacing references to constants with immediate operands 800
538. Replacing short unsigned displacements with long signed displacements 800
539. A program fragment needing reorganization 801
540. A program fragment after reorganization 801
541. Reorganizing a program to minimize base registers 801
542. Incorrect implied reference to a different control section 804
543. Correct implied reference to a different control section 804
544. USING Table with two entries 805
545. Main program and subroutine in one assembly 805
546. Main program, subroutine, and common section in one assembly 806
547. Resuming control sections 806
548. Main program and subroutine in one assembly, multiple CSects 807
549. Statements with Location Counter discontinuities 808
550. Technique for rounding the length of a CSECT 809
551. Rearrangement of source groups by LOCTR 811
552. Simple example of LOCTR (1) 811
553. Simple example of LOCTR (2) 812
554. Simple example of LOCTR (3) 812
555. A program fragment using LOCTR for reorganization 812
556. Organizing a program to minimize addressability problems 813
557. Organizing a program to minimize addressability problems 813
558. Simple example of LOCTR (4) 814
559. Example of unexpected LOCTR behavior (1) 814
560. Example of unexpected LOCTR behavior (2) 815
561. Calling ShftRt as an external routine 819
562. ShftRt subroutine as a separate assembly 819
563. External references using relative branch instructions 820
564. Using WXTRN to test whether a routine was linked 820
565. Calling ShftRt as an external routine 821
566. ShftRt subroutine in a different CSect 822

567. Main program with ENTRY for data . . . 822
568. Subroutine using EXTRN to reference data . . . 822
569. Subroutine using EXTRN and adcons to reference data . . . 823
570. Subroutine with entries for two similar functions . . . 824
571. Subroutine with two similar functions and some common code . . . 824
572. Sample assembly with external symbols . . . 825
573. External symbol dictionary from sample assembly . . . 825
574. Program assembled with different SECTALGN options . . . 827
575. Example of ESD listings with different SECTALGN options . . . 827
576. Assigning RMODE and AMODE to a section name . . . 828
577. ESD showing RMODE and AMODE of section names . . . 829
578. Example of two source modules to be linked . . . 834
579. Sketch of object module from source module 1 . . . 834
580. Sketch of object module from source module 2 . . . 835
581. Composite ESD after reading first object module . . . 835
582. Composite ESD after loading second object module . . . 836
583. Composite ESD after assigning memory addresses . . . 836
584. Memory layout of loaded program . . . 837
585. Sample DXD declarations . . . 838
586. External dummy section declaration . . . 838
587. Referencing external dummy items with Q-cons . . . 838
588. External dummy items in ESD listing . . . 839
589. Separate DXD declaration . . . 839
590. Example of a completed External Dummy Section . . . 839
591. Retrieving an External Dummy Section item . . . 840
592. PL/I technique for loading Pseudo Registers . . . 840
593. ESDID Translation Table entry for an incoming symbol . . . 842
594. A typical load-time CESD entry . . . 842
595. Composite ESD after assigning load module addresses . . . 845
596. Sketch of a load module . . . 846
597. A load module after loading . . . 847
598. Sketch of program object structure . . . 849
599. Sample program assembled with the GOFF option . . . 849
600. ESD from program assembled with the GOFF option . . . 850
601. Assigning AMODE to an entry symbol . . . 851
602. ESD showing AMODE assigned to entry and external symbols . . . 851
603. Sample program defining two Sections and three Classes . . . 851
604. Assignment of instructions and data into elements . . . 851
605. Assembly listing for sample program . . . 852
606. External symbol dictionary for sample program . . . 852
607. Example of declaring parts in a GOFF Class . . . 853
608. ESD for parts in a GOFF Class . . . 853
609. Sketch of virtual memory . . . 856
610. Sample program defining two Sections and three Classes . . . 856
611. Sketch of classes in virtual memory . . . 857
612. System z PSW showing addressing-mode bits . . . 858
613. Important addressing mode bits for BASSM . . . 859
614. BASSM setting of first-operand register for 24-, 31-, and 64-bit addressing modes . . . 859
615. Sketch of residence and addressing modes . . . 863
616. Example showing why LLGT/LLGTR are necessary . . . 864
617. Example showing why LLGTR is important . . . 865
618. Example of a dummy control section . . . 873
619. Example using a dummy control section . . . 873
620. USING Table with two entries, one for a dummy section . . . 874
621. Object code from references to a dummy control section . . . 874
622. Example using a dummy control section . . . 874
623. A poor method for describing two instances of a record . . . 875
624. A better record description with a DSECT . . . 876
625. Ordinary USING statement syntax . . . 876
626. Copying a field from Old record to New . . . 877
627. Incorrect addressing with ordinary USING . . . 878
628. Correct but awkward addressing with ordinary USING . . . 878

629. Manual coding of base and displacement for a large DSECT . . . 879
630. Labeled USING statement syntax . . . 882
631. Qualified symbol syntax . . . 882
632. Examples of qualifier definitions . . . 882
633. Copying a field with Labeled USINGs . . . 883
634. DROP statement for Labeled USING . . . 883
635. Concurrently active Ordinary and Labeled USINGs . . . 884
636. Dummy control section for record address . . . 885
637. Improved definition of a record description . . . 885
638. Mapping a substructure with a second DSECT . . . 886
639. Dependent USING statement syntax . . . 886
640. Anchoring an internal DSECT with a Dependent USING . . . 887
641. Outer DSECT with two nested DSECTs . . . 887
642. Assembler listing of multiple Dependent USINGs and DSECTs . . . 888
643. Three independent data structures with one base register . . . 888
644. Defining DSECTs for three independent data structures . . . 889
645. Defining a mapping of three independent but contiguous data structures . . . 889
646. Example of a message-skeleton CSECT . . . 889
647. Example of mapping a CSECT as though it is a DSECT . . . 890
648. Labeled Dependent USING statement syntax . . . 891
649. Nesting two identical structures within a third . . . 891
650. Addressing two nested DSECTs with Labeled Dependent USINGs . . . 893
651. Data in nested DSECTs addressed with Labeled Dependent USINGs . . . 893
652. Multiply-Nested Data Structures . . . 893
653. Doubly Nested DSECT definitions . . . 894
654. Addressing doubly nested DSECT definitions . . . 894
655. Using the Labeled Dependent USINGs to move data . . . 895
656. Addressing two DCBs with ordinary USINGs . . . 896
657. Addressing instructions and DCBs with one register . . . 897
658. Define a personnel-file record . . . 898
659. Employee-record Person DSECT . . . 898
660. Employee-record Date DSECT . . . 898
661. Employee-record Address DSECT . . . 899
662. Employee-record Phone DSECT . . . 899
663. DSECT nesting in an employee record . . . 900
664. Anchoring various DSECTs within Employee record . . . 901
665. Manipulating fields within an Employee record . . . 901
666. Addressing DSECTs within Employee record with ordinary USINGs . . . 901
667. Comparing dates of birth in Employee record . . . 902
668. Comparing date fields in different parts of an Employee record . . . 902
669. Copying addresses with an Employee Record . . . 903
670. Example of a one-dimensional array of halfwords . . . 908
671. Sum of array elements with known subscript bounds . . . 908
672. Sum of array elements with unknown subscript bounds . . . 909
673. Typical arrangement of elements of a matrix . . . 910
674. Storing an array in column order . . . 910
675. Storing an array in row order . . . 910
676. Retrieving a specified element of an array . . . 911
677. Retrieving a specified element of an array efficiently . . . 912
678. Searching for a matching table entry . . . 914
679. Searching for a table entry mapped by a DSECT . . . 915
680. USING Table with two entries, one for a DSECT . . . 915
681. Creating a table of addresses . . . 917
682. Creating a better table of addresses . . . 917
683. Creating a table of addresses at assembly time . . . 918
684. Example of a binary search . . . 921
685. A stack growing toward higher addresses . . . 924
686. A stack implemented as an array . . . 924
687. Pushing a data item onto a stack . . . 924
688. Adding top two elements of a stack . . . 925
689. A stack growing toward lower addresses . . . 925
690. Add top two elements of a stack . . . 925

691. Sketch of a linked list 927
692. Inserting an element into a linked list 928
693. Example of inserting an element into a linked list 928
694. DSECT describing a list element 928
695. Mapping multiple list elements with Labeled USINGs 928
696. Deleting an element from a linked list 929
697. Example of deleting an element from a linked list 929
698. Example of deleting an element from a linked list 929
699. Defining a free storage list as an array 930
700. Initializing a free storage list as an array 930
701. Example of a list anchor 930
702. DSECT mapping a list anchor 930
703. Defining an anchor for a working list 931
704. Moving a list element from the FSL to the working list 931
705. A two-dimensional array to implement a linked list 932
706. Initializing a two-dimensional array implementing a linked list 932
707. Structure of a queue element 934
708. A queue with several elements 934
709. DSECT structure of a typical queue element 934
710. An element to be inserted into a queue 935
711. A queue after insertion of a new element 935
712. Instructions to insert a new queue element 935
713. Insert a new list element with ordinary USINGs 936
714. Ordinary-USING Code to Insert a New List Element 936
715. Labeled USING example: inserting a new queue element 936
716. Node of a binary tree 937
717. DSECT structure of a typical tree element 937
718. Three nodes of a binary tree 938
719. A growing binary tree with seven nodes 938
720. Entering a new node in a binary tree 939
721. Retrieving data from a binary tree 940
722. Example of a binary tree of 7 elements 940
723. Example of searching a hash table 942
724. Example of searching a hash table 943
725. Sample macro invocation, Standard form 951
726. Generated statements from an OPEN macro 952
727. Sample macro invocation using List form 952
728. Generated statements from a List form OPEN macro 953
729. Sample macro invocation using Execute form 953
730. Generated statements from an Execute form OPEN macro 953
731. Sample macro invocation using empty List form 953
732. Generated instructions from empty List form 953
733. Sample macro invocation using Execute form 953
734. Generated statements from an Execute form OPEN macro 953
735. Another macro invocation using Execute form and same List form 954
736. An R-Type macro invocation generating an argument in a register 954
737. Generated statements from R-Type macro 954
738. A macro invocation with arguments in registers 954
739. Generated statements from a Standard-form macro with arguments in registers 954
740. A Standard macro invocation specifying MODE=31 955
741. Generated statements from a Standard-for macro with MODE=31 955
742. Example of a mixed-case positional macro argument 955
743. Example of mixed-case keyword macro arguments 955
744. Example of mixed-case keyword macro arguments 955
745. Sample ABEND macro 956
746. Generated statements from an ABEND macro 957
747. Sample R=type GETMAIN request 958
748. Expansion of a sample R-type GETMAIN request 959
749. Expansion of a sample VRU-type GETMAIN request 959
750. Example of an R-type FREEMAIN macro 960
751. Sample STORAGE OBTAIN request 960
752. Example of a STORAGE OBTAIN macro expansion 961

753. Sample STORAGE RELEASE request . . . 961
754. Example of a STORAGE RELEASE macro expansion . . . 961
755. A Data Set with records you want to read . . . 962
756. You submitted a job with a program to read the records . . . 963
757. Your program, loaded into memory before execution . . . 963
758. Your program after executing the OPEN macro . . . 964
759. Your program after executing the GET macro . . . 965
760. Your program after executing the CLOSE macro . . . 965
761. Example of typical DCB parameters . . . 968
762. Unblocked and blocked F-type record and block formats . . . 968
763. Unblocked and blocked V-type record and block formats . . . 969
764. U-type block formats . . . 969
765. Completion of a DCB during OPEN processing . . . 969
766. DCBD operands . . . 970
767. DCBD operands . . . 970
768. Using IHADCB to map two different DCBs simultaneously . . . 970
769. A complete sample program . . . 972
770. Instruction cycle with interruptions . . . 973
771. Establishing a program interruption exit . . . 973
772. Expansion of an ESPIE macro establishing a program interruption exit . . . 974
773. Terminating a program interruption exit . . . 974
774. Expansion of an ESPIE macro terminating a program interruption exit . . . 974
775. ESA/390-mode old PSW in EPIE . . . 975
776. Sketch of interruption handling control flow . . . 977
777. A simple ESTAE macro. . . . 979
778. Skeleton form of a reenterable program . . . 985
779. I/O macros in a reenterable program . . . 986
780. Assembly listing for a simple reenterable program . . . 986
781. Example of a reenterable, recursive routine . . . 990
782. Assembly listing of the reenterable recursive routine . . . 991

Tables

1. Binary, decimal, and hexadecimal . . . 18
2. Multiples of powers of sixteen (part 1 of 2) . . . 20
3. Multiples of powers of sixteen (part 2 of 2) . . . 21
4. Examples of two's complement representation . . . 29
5. Examples of sign extension . . . 31
6. RR-type instruction format . . . 52
7. RX-type and RS-type instruction format . . . 52
8. SI-type instruction format . . . 52
9. SS-type instruction format . . . 53
10. Instruction Length Code and instruction types . . . 54
11. General instruction classifications . . . 55
12. Punched-card image of a RETURN statement . . . 78
13. Assembler Language EBCDIC character representation . . . 87
14. Differences between Assembler Language and high-level language symbols . . . 94
15. Expressions with absolute and relocatable terms . . . 99
16. Typical RR-type instructions . . . 106
17. RR-type instruction . . . 107
18. Typical RX-type instructions . . . 108
19. RX-type instruction . . . 108
20. Operands of RX-type instructions . . . 109
21. Typical RS- and SI-type instructions . . . 111
22. Typical RS-type instruction . . . 111
23. Operands of RS-type instructions . . . 112
24. Typical SI-type instruction . . . 112
25. Operands of SI-type instructions . . . 112

26. Typical SS-type instructions 113
27. Typical type SS-1 instruction with one length field 113
28. Operands of type SS-1 single-length instructions 113
29. Typical type SS-2 instruction with two length fields 114
30. Operands of type SS-2 two-length instructions 114
31. Examples of truncated and padded constants 152
32. Truncation/padding rules for some DC operands 153
33. Truncation and padding rules for some DC operands with extended types 158
34. Load/Store instructions for 32-bit general registers 179
35. Format of an RX-type instruction 179
36. Multiple load/store instructions for 32-bit general registers 180
37. RS-type instruction format 180
38. Halfword load/store instructions for 32-bit general registers 182
39. Character insert/store instructions for 32-bit general registers 184
40. Insert/Store characters under mask instructions for 32-bit general registers 185
41. RS-type instruction format for ICM and STCM 186
42. CC settings after ICM instruction 187
43. Register/register instructions for 32-bit general registers 187
44. Action of five RR-type general register instructions 188
45. Condition Code settings 188
46. Register/storage instructions for 64-bit general registers 189
47. RXY-type instruction format 190
48. RSY-type instruction format. 190
49. RRE-type instruction format 192
50. Register/register instructions for 64-bit general registers 192
51. Action of five RR-type 64-bit general register instructions 192
52. Load and Test instructions 194
53. Register/register instructions for 64-bit general registers 194
54. Action of 32-bit-to-64-bit general register instructions 194
55. Other general register load instructions 196
56. Summary of instructions discussed in this section 200
57. BCR instruction 205
58. BC instruction 205
59. Mask bits and corresponding CC values 205
60. CNOP operands 208
61. Extended branch mnemonics and their branch mask values 210
62. Frequently used add and subtract instructions 216
63. CC settings for arithmetic add and subtract instructions 217
64. Arithmetic compare instructions 222
65. CC settings after arithmetic comparisons 222
66. Logical arithmetic instructions 224
67. CC settings for logical add and subtract instructions 224
68. CC indications for logical addition and subtraction 225
69. CC settings after logical addition 228
70. CC settings after logical subtraction 228
71. Logical arithmetic instructions with carry/borrow 228
72. Instructions for mixed-length operands 230
73. Arithmetic compare instructions 232
74. CC settings after logical comparisons 232
75. IPM and SPM instructions 234
76. Program Mask bits 235
77. Summary of instructions discussed in this section 236
78. General register shift instructions 242
79. RS-type shift instruction 243
80. RSY-type instruction format 243
81. CC settings for arithmetic shift instructions 252
82. Summary of shift instructions discussed in this section 260
83. Binary integer multiply instructions 264
84. Double-length arithmetic multiply instructions 265
85. Single-length arithmetic multiply instructions 268
86. Logical multiply instructions 270
87. Binary divide instructions 274

88. Arithmetic divide instructions . . . 275
89. Binary divide instructions . . . 279
90. Summary of multiply instructions discussed in this section . . . 283
91. Summary of divide instructions discussed in this section . . . 283
92. Logical operations involving general registers . . . 288
93. CC settings by logical instructions . . . 289
94. Summary of the logical operations AND, OR, XOR . . . 297
95. Logical-operation instructions discussed in this section . . . 297
96. Format of RXY- and RSY-type instructions . . . 302
97. Format of R-I instructions with 16-bit immediate operands . . . 305
98. Format of R-I instructions with 32-bit immediate operands . . . 305
99. PSW addressing-mode bits . . . 309
100. Load Address instructions . . . 309
101. Load Address instructions described in this section . . . 314
102. RI-type instruction . . . 317
103. RIL-type instruction . . . 317
104. Insert-Immediate instructions . . . 318
105. Load and insert instructions with immediate operands . . . 319
106. Arithmetic-immediate add and subtract instructions . . . 321
107. Arithmetic-immediate compare instructions . . . 322
108. Arithmetic-immediate multiply instructions . . . 322
109. AND-immediate instructions . . . 323
110. OR-immediate instructions . . . 324
111. XOR-immediate instructions . . . 324
112. Load and insert instructions with immediate operands . . . 326
113. Arithmetic instructions with immediate operands . . . 326
114. Logical instructions with immediate operands . . . 326
115. Format of the BRC instruction . . . 329
116. Format of the BRCL instruction . . . 329
117. Extended branch relative on condition mnemonics and their branch mask values . . . 330
118. Branch on count instructions . . . 334
119. Extended mnemonics for branch relative on count instructions . . . 334
120. Branch on index instructions . . . 341
121. RS-type BXH and BXLE instructions . . . 341
122. RSY-type BXHG and BXLEG instructions . . . 341
123. RSI-type BRXH and BRXLE instructions . . . 341
124. RIE-type BRXHG and BRXLG instructions . . . 341
125. Extended mnemonics for branch relative on index instructions . . . 343
126. Branch relative on condition instructions . . . 349
127. Branch instructions for loop control . . . 349
128. SI-type instruction format . . . 352
129. SIY-type instruction format . . . 352
130. SI-type instruction actions . . . 353
131. Move Immediate instructions . . . 353
132. Logical Storage-Immediate instructions . . . 353
133. CC settings by SI-type logical instructions . . . 354
134. Compare Immediate instructions . . . 354
135. CC settings after CLI instruction . . . 355
136. Storage-Immediate instructions . . . 356
137. CC settings after TM instruction . . . 356
138. Storage-Immediate instructions . . . 363
139. Basic character-handling instructions . . . 365
140. Format of single-length SS-type instructions . . . 365
141. Instruction types and operand formats . . . 367
142. SS-type instructions with explicit length . . . 367
143. SS-type instructions with implied length . . . 368
144. Determining the Length Specification Byte . . . 370
145. MVCOS instruction . . . 374
146. SSF instruction format used for the MVCOS instruction . . . 374
147. Condition Code settings for TRT and TRTR instructions . . . 384
148. Execute instructions . . . 389
149. Modifiable portions of typical EX target instructions . . . 394

150. Operands of single-length SS-type instructions 396
151. Basic instructions for data in storage 397
152. Basic character-handling instructions using padding characters 404
153. CC settings after MVCL 405
154. CC settings after CLCL 407
155. Format of MVCLE and CLCLE instructions 410
156. CC settings after MVCLE 412
157. CC settings after CLCLE 414
158. Character-handling instructions for terminated strings 415
159. Format of RRE-type instructions 415
160. CC settings for SRST instruction 416
161. CC settings for MVST instruction 417
162. CC settings for CLST instruction 419
163. CC settings for TRE instruction 421
164. Compare Until Substring Equal instruction 423
165. Condition Code settings by CUSE 424
166. Results of examples using the CUSE instruction 424
167. Extended instructions for character data 425
168. Punched paper tape encodings with values 00-0F 429
169. Punched paper tape encodings with values 10-1F 429
170. Old six-bit BCD character representation 430
171. Sample EBCDIC characters with varying code points among code pages 431
172. 7-bit ASCII character representation 433
173. Japanese DBCS assignments 435
174. DBCS encoding 435
175. Sample Unicode assignments 439
176. Unicode string instructions 441
177. CC settings for SRSTU instruction 441
178. CC settings after MVCLU 442
179. CC settings after CLCLU 443
180. RRE-type instruction 443
181. RRF-format instruction with an optional operand 444
182. Unicode translate instructions 444
183. Arguments and translate tables for TRxx instructions 445
184. Registers used by TRxx instructions 445
185. Condition Code settings for TRxx instructions 445
186. Unicode format conversion instructions 448
187. CC settings after Unicode format conversion instructions 448
188. Translate and Test Extended instructions 450
189. Function-code table sizes for TRTE, TRTRE 450
190. Condition code settings for TRTE, TRTRE 451
191. Byte-reversing load and store instructions 453
192. Extended instructions for Unicode data 456
193. Unicode-based translate instructions 456
194. Unicode format conversion instructions 456
195. Summary of byte-reversing instructions 456
196. Basic packed and zoned decimal instructions 460
197. Examples of zoned decimal data 462
198. Punched-card image of two numbers, +12345 and −67890 463
199. Examples of packed decimal data 466
200. Format of two-length SS-type instructions 469
201. Operands of two-length SS-type instructions 470
202. Format of PKA and PKU instructions 478
203. Format of UNPKA and UNPKU instructions 479
204. CC settings after UNPKA, UNPKU instructions 480
205. Instructions for moving numeric and zone digits 483
206. Instructions for packing and unpacking data 483
207. CC settings for decimal addition and subtraction 486
208. CC setting after decimal comparison 488
209. Packed decimal arithmetic instructions 497
210. Operand formats for TP instruction 498
211. Format of the TP instruction 498

212. CC settings for the TP instruction . . . 498
213. CC settings by the ZAP, AP, and SP instructions . . . 499
214. CC setting by the CP instruction . . . 504
215. Format of the SRP instruction . . . 511
216. Summary of decimal instruction behavior . . . 530
217. Instructions used for converting and formatting packed decimal . . . 532
218. Format of the ED and EDMK instructions . . . 536
219. CC settings after ED, EDMK . . . 544
220. ED and EDMK treatment of pattern characters . . . 547
221. Basic floating-point instructions . . . 568
222. Instructions copying data between FPRs . . . 568
223. Floating-point Load Zero instructions . . . 569
224. Instructions moving data between FPRs and GPRs . . . 570
225. Copy Sign instruction . . . 570
226. Basic Load/Store instructions for floating-point operands . . . 571
227. Instructions moving operands between GPRs and FPRs . . . 571
228. Hexadecimal floating-point data representations . . . 587
229. Unnormalized and normalized short hexadecimal floating-point numbers . . . 588
230. Short hexadecimal floating-point numbers . . . 588
231. Long hexadecimal floating-point numbers . . . 589
232. Extended hexadecimal floating-point numbers . . . 589
233. Assembled hexadecimal floating-point constants . . . 591
234. Hex floating-point constants with decimal exponents . . . 591
235. Length-modified hexadecimal floating-point constants . . . 592
236. Hexadecimal floating-point constants with modifiers . . . 594
237. Hexadecimal floating-point rounding modes with subtype H . . . 595
238. Symbolic hexadecimal floating-point constants . . . 596
239. "Difficult" hexadecimal floating-point conversion values . . . 596
240. Data-moving hexadecimal floating-point instructions . . . 597
241. Hexadecimal floating-point Multiply instructions . . . 599
242. Summary of hexadecimal floating-point multiplication results . . . 601
243. Hexadecimal floating-point Divide instructions . . . 603
244. Hexadecimal floating-point Halve instructions . . . 604
245. Hexadecimal floating-point Add/Subtract instructions . . . 606
246. Hexadecimal floating-point Compare instructions . . . 615
247. CC settings for hexadecimal floating-point comparison . . . 616
248. Hexadecimal floating-point Round instructions . . . 616
249. Hexadecimal floating-point Load Lengthened instructions . . . 619
250. Hexadecimal floating-point FPR/GPR conversion instructions . . . 620
251. Format of HFP to fixed binary instructions . . . 621
252. Rounding modifiers for HFP-to-binary conversion . . . 621
253. CC settings for HFP-to-binary conversion . . . 622
254. Instructions moving/converting binary and hexadecimal floating-point operands . . . 622
255. Hexadecimal floating-point instructions generating floating-point integers . . . 625
256. Hexadecimal floating-point Square Root instructions . . . 627
257. Hexadecimal floating-point Multiply and add/subtract instructions . . . 627
258. Format of RRF-type HFP multiply and add/subtract instructions . . . 628
259. Format of RXF-type multiply and add/subtract instructions . . . 628
260. Hexadecimal floating-point Move/Test instructions . . . 631
261. Hexadecimal floating-point Multiply instructions . . . 631
262. Hexadecimal floating-point Divide instructions . . . 631
263. Hexadecimal floating-point Add, Subtract, and Compare instructions . . . 631
264. Hexadecimal floating-point Round instructions . . . 632
265. Hexadecimal floating-point Lengthening instructions . . . 632
266. Convert hexadecimal floating-point to binary instructions . . . 632
267. Convert binary to hexadecimal floating-point instructions . . . 632
268. Form hexadecimal floating-point integer instructions . . . 632
269. Hexadecimal floating-point Square Root instructions . . . 632
270. Hexadecimal floating-point Multiply-Add/Subtract instructions . . . 633
271. Binary floating-point data representations . . . 638
272. Examples of short-precision binary floating-point normal values . . . 640
273. Examples of short-precision binary floating-point denormalized values . . . 640

274. Examples of short-precision binary floating-point special values 641
275. Nominal-value operands for binary floating-point special values 644
276. Assembled binary floating-point special-value constants 644
277. Minimum bit lengths for binary floating-point constants 645
278. Binary floating-point DXC values 650
279. Binary floating-point FPC register control instructions 651
280. Invalid operation binary floating-point exception 652
281. Divide by zero binary floating-point exception 652
282. Exponent overflow binary floating-point exception 652
283. Exponent underflow binary floating-point exception 652
284. Inexact result binary floating-point exception 652
285. BFP overflow/underflow scale factors 653
286. Binary floating-point Test Data Class instructions 654
287. Test Data Class second-operand bits 654
288. Test Data Class second-operand test-bit/tested-value correspondence 654
289. Binary floating-point RR-type data movement instructions 655
290. CC settings for BFP data movement instructions 656
291. Binary floating-point Multiply instructions 657
292. Binary floating-point Divide instructions 659
293. Binary floating-point Add and Subtract instructions 661
294. CC settings after BFP add/subtract instructions 661
295. Binary floating-point Compare instructions 662
296. CC settings for BFP comparisons 662
297. Binary floating-point Compare and Signal instructions 663
298. Binary floating-point Round instructions 664
299. Binary floating-point Lengthening instructions 665
300. Binary integer to binary floating-point conversion instructions 666
301. Binary floating-point to integer conversion instructions 666
302. Format of BFP Convert To Fixed instructions 666
303. Rounding modifier for BFP convert to fixed instructions 667
304. CC settings after convert to binary instructions 667
305. Load floating-point integer instructions 668
306. Rounding mode modifiers for BFP load integer instructions 669
307. Binary floating-point Divide to Integer instructions 669
308. Format of BFP Divide to Integer instructions 669
309. CC settings after divide to integer instructions 670
310. Binary floating-point Square Root instructions 671
311. Binary floating-point Multiply and Add/Subtract instructions 672
312. Summary of binary floating-point instructions with uniform operand lengths 673
313. Binary floating-point Multiply instructions 674
314. Binary floating-point Round instructions 674
315. Binary floating-point Lengthening instructions 674
316. Convert binary floating-point to binary integer instructions 675
317. Convert binary integer to binary floating-point instructions 675
318. Summary of binary floating-point operations and exceptions 675
319. Decimal floating-point data representations 684
320. Declet encoding for BCD digits 685
321. Converting decimal floating-point declets to BCD digits 686
322. First five bits of special-values Combination Field 687
323. First 5 bits of finite-value Combination Field 688
324. Properties of decimal floating-point representations 689
325. Assembled decimal floating-point special-value constants 690
326. Examples of decimal floating-point short precision zeros 691
327. Assembler rounding-mode suffixes for DFP constants 691
328. Decimal floating-point Test Data Class instructions 693
329. DFP Test Data Class second-operand bits 694
330. Test Data Class test-bit vs. tested-class correspondence 694
331. Example of DFP rounding modes 696
332. Preferred quanta for some decimal floating-point operations 698
333. Decimal floating-point additional DXC value 699
334. Decimal floating-point quantum exception 699
335. Decimal floating-point scale factors for exponent spills 699

336. Copy Sign instruction . . . 700
337. Instructions moving data between FPRs and GPRs . . . 700
338. Instructions copying data between FPRs . . . 700
339. Decimal floating-point basic arithmetic instructions . . . 701
340. Format of DFP arithmetic instructions . . . 701
341. Format of DFP arithmetic instructions with rounding mask . . . 701
342. Instruction-specific rounding mask values . . . 702
343. CC settings for Add/Subtract instructions . . . 704
344. CC settings for Compare instructions . . . 705
345. Decimal floating-point Compare instructions . . . 705
346. Decimal floating-point Compare and Signal instructions . . . 705
347. Decimal floating-point Compare Biased Exponent instructions . . . 706
348. CC settings for Compare Biased Exponent instructions . . . 706
349. Decimal floating-point convert to/from fixed binary instructions . . . 707
350. Format of Convert to Fixed Binary instructions . . . 708
351. Format of Convert to Fixed Binary instructions . . . 708
352. CC settings for Convert to Fixed instructions . . . 708
353. Decimal floating-point convert to/from signed packed decimal instructions . . . 709
354. Format of Convert to Signed Packed instructions . . . 709
355. Decimal floating-point convert to/from unsigned packed decimal instructions . . . 711
356. Instructions converting between decimal floating-point and zoned decimal . . . 712
357. Format of DFP/zoned decimal conversion instructions . . . 712
358. Condition Code settings for Convert to Zoned . . . 713
359. Decimal floating-point Load and Test instructions . . . 715
360. CC setting after DFP Load and Test instructions . . . 715
361. Instructions copying/complementing data between FPRs . . . 715
362. Decimal floating-point Load Floating-point Integer instructions . . . 715
363. Format of Load FP Integer instructions . . . 715
364. Decimal floating-point Load Lengthened instructions . . . 716
365. Load Lengthened special operand control mask . . . 716
366. Decimal floating-point rounding/lengthening instructions . . . 717
367. Decimal floating-point Set Rounding Mode instruction . . . 718
368. Decimal floating-point Insert/Extract Biased Exponent instructions . . . 719
369. Extracted Biased Exponent for DFP special values . . . 719
370. DFP Insert Biased Exponent results . . . 720
371. Decimal floating-point Extract Significance instructions . . . 720
372. Decimal floating-point Shift Significand instructions . . . 721
373. Format of DFP shift instructions . . . 721
374. Decimal floating-point Quantize instructions . . . 722
375. Format of decimal floating-point Quantize instructions . . . 722
376. Decimal floating-point Reround instructions . . . 724
377. Decimal floating-point Test Data Group instructions . . . 726
378. Test Data Group second-operand bits . . . 726
379. DFP Test Data Class and Test Data Group instructions . . . 732
380. DFP Arithmetic and related instructions . . . 732
381. DFP length and type conversion instructions . . . 732
382. DFP rounding and lengthening instructions . . . 732
383. DFP data-loading instructions . . . 733
384. Instructions copying between FPRs and GPRs . . . 733
385. Instruction setting decimal rounding mode . . . 733
386. Non-canonical declets . . . 733
387. Summary of System z floating-point representations . . . 739
388. Adding 0.1 in hexadecimal, binary, and decimal floating-point . . . 741
389. Exception behavior for hexadecimal floating-point . . . 741
390. Exception behavior for binary and decimal floating-point . . . 741
391. Length modifiers of floating-point constants . . . 742
392. Assembler rounding-mode suffixes for floating-point constants . . . 742
393. Internal precision required for faithful In-Out conversion . . . 744
394. Decimal precision required for faithful Out-In conversion . . . 744
395. Perform Floating-Point Operation instruction . . . 745
396. Laws of real and realistic arithmetic . . . 746
397. Examples of hexadecimal floating-point pseudo-zeros . . . 748

398. Examples of other floating-point representations 750
399. Equivalent decimal and floating-point precisions 751
400. Branch and Save instructions 758
401. Standard (Format-0) Save Area 770
402. Standard Format-4 save area 773
403. Standard Format-5 save area 775
404. AMODE values 827
405. RMODE values 828
406. Default AMODE and RMODE values 828
407. Valid combinations of AMODE and RMODE values 828
408. Differences in linking COMMONs and External dummy items 841
409. ESD symbol search types 841
410. Matching existing CESD SD symbol to incoming symbols 843
411. Matching existing CESD LD symbol to incoming symbols 843
412. Matching existing CESD CM symbol to incoming symbols 843
413. Matching existing CESD ER symbol to incoming symbols 844
414. Matching existing CESD ER symbol to incoming symbols 844
415. Comparing load modules and program objects 855
416. Instructions to change addressing mode 858
417. CC settings for TAM instruction 858
418. PSW addressing-mode bits 858
419. BASSM actions summary 860
420. Operation of BSM instruction 860
421. BSM actions summary 861
422. Instruction pairs for call/return with possible AMODE change 862
423. Calling among addressing modes within an assembly 863
424. LLGT and LLGTR instructions 863
425. Symbol table entries for DSECT symbols 873
426. Summary of USING Statements 905
427. Summary of DROP Statement Behaviors 905
428. Example of a non-homogeneous array 914
429. Array addressing with a table of addresses 917
430. Example of an address table's contents 918
431. Supervisor and Program Call instructions 950
432. SVC instruction 950
433. PC instruction format 951
433. Program Call instruction 951
434. GETMAIN request options 958
435. FREEMAIN request options 960
436. Comparing QSAM and BSAM 966
437. Partial contents of Extended Program Interruption Element (EPIE) 975
438. Hexadecimal, decimal, and binary 995
439. Hexadecimal Addition Table 996
440. Hexadecimal Multiplication Table 996
441. Integer powers of 2 997
442. Integer powers of 2 998
443. Multiples of powers of sixteen (part 1 of 2) 1000
444. Multiples of powers of sixteen (part 2 of 2) 1000
445. Powers of 10 expressed in hexadecimal 1001
446. Assembler Language EBCDIC character representation 1012
447. 7-bit ASCII character representation 1013
448. High Level Assembler DC-Statement Constant Types 1014
449. ASCII Character Representation 1077
450. Examples of different types of integer division 1129
451. Comparing five binary floating-point operands 1225

Foreword

```
FFFFFFFFFFFF  WW          WW
FFFFFFFFFFFF  WW          WW
FF            WW          WW
FF            WW          WW
FF            WW          WW
FFFFFFFF      WW          WW
FFFFFFFF      WW    WW    WW
FF            WW   WWWW   WW
FF            WW  WW  WW  WW
FF            WWWW      WWWW
FF            WWW        WWW
FF            WW          WW
```

Outline and Overview

We will survey many aspects of Assembler Language programming on System z processors.

Chapters I-IV cover basic material needed for almost all programs.

- Chapter I introduces some notation we'll use, and discusses the important topics of binary and hexadecimal number representations and arithmetic, and conversion among number representations.
- Chapter II introduces the “Central Processing Unit” or CPU. We'll survey central memory, the registers you'll use in your programs, and the Program Status Word (PSW). Then we'll look at some basic types of instructions and their operation codes, and see how they refer to data in memory.
- Chapter III describes basic properties of the Assembler Language, including symbols, self-defining terms, and expression evaluation. Then we will see how to write Assembler Language statements and their components. Last, we discuss the key concept of addressability and the important USING statement.
- Chapter IV describes methods for defining often-used data types, and techniques for organizing data items in your programs.
- The six sections of Chapter V discuss basic instructions, emphasizing those that operate on data in the general registers, and the important “conditional branch” instructions.
- Chapter VI considers addressing techniques, loops and other iterative processes, and “immediate” instructions containing useful operands.
- Chapter VII discusses bit and character data and techniques for handling them.
- Chapter VIII examines the packed and zoned decimal data representations, instructions for packed decimal arithmetic, and for conversion between those representations and EBCDIC characters.
- Chapter IX describes general concepts of floating-point arithmetic and the three floating-point representations supported by System z: hexadecimal, binary, and decimal, and instructions for manipulating data in and among each of the representations. It concludes with a summary of important differences between floating-point and mathematicians' “real” arithmetics.
- Chapter X discusses large programs and modularization techniques such as subroutines and common linkage conventions, how to combine separately assembled (or compiled) routines into a single executable program, and how to change addressing modes.
- Chapter XI describes the powerful “Dummy Control Section” and the enhanced USING statements, and shows how to apply them to several basic data structures.

- Chapter XII introduces common techniques for accessing operating system services, basics of exception handling, and uses of reenterability and recursion.
- Appendix A contains reference and conversion tables.
- Appendix B describes a set of useful macro instructions that handle simple input, output, conversion, and display operations.

Programming Environments

Every programming language must eventually deal with the environments under which the programs will be run. While we will see many examples of program segments, we will defer complete programs until later sections.

I assume your programs will execute on one of z/OS™, z/VM™, or z/VSE™. I have purposely omitted discussion of z/Linux™, because Assembler Language is little used in that environment.

If you like, browse the solutions to the Programming Problems: these are complete programs that have been executed successfully, and produce what I believe are correct answers. The simple conventions used here for communicating with the Operating System's Supervisor are described in "Appendix B: Simple I/O Macros" on page 1015; these may be augmented or replaced as desired.

The conventions and procedures needed to execute an Assembler Language program in your computing environment should be locally available to you.

Levels of Difficulty (*)

This material varies in depth and detail. Where a detailed portion can be skipped with no loss of continuity, the heading is tagged with a parenthesized asterisk (*), as in the heading just above.

Exercises and Programming Problems

Exercises and programming problems appear throughout. Some are integral to the material, while others explore interesting sidelines. Exercises and programming problems are rated in order of estimated difficulty from 1 to 5; the most useful or illustrative exercises are tagged with a plus (+), and are strongly recommended.

In all cases, the exercises and programming problems are important.

Some Personal Observations

1. Some exercises ask you to find what is wrong with a statement or instruction sequence. While it may be poor style (or manners) to show coding errors, I feel justified in doing so on two grounds: pedagogical value and self-defense.
   - First, it helps to see wrong or poor ways to do something, as well as correct or better ways.
   - Second, some programs may be written by people who learned from examples containing errors — and their programs will be processing my bills, checking my tax returns, and calculating my bank balance. I want your programs — and theirs — to be as safe, correct, and reliable as possible.

   I trust you will understand. I am of course willing to have you point out *my* errors. If you find any, please let me know so I can correct them.
2. This is not intended to be a cookbook. I have tried to give not just occasional recipes for doing some basic tasks, but a view of some underlying processor structures and the language closest to the processor, Assembler Language. You may have already been introduced to programming a computer using a "higher-level" language, and are probably familiar with concepts such as loops and conditional branching. Because the internal structures of computers have many similarities, I sometimes try to point out not only what a particular

instruction does, but also why it does it that way. Learning to program other processors will then be a comfortable extension of the concepts and techniques you learned here.

3. This book is too large[1] to be used as a text for a programming class of normal length. I expect that most instructors will use those portions most useful for their selection of topics; other portions may have information that can be sampled as desired.

   I assume you are interested mainly in writing “application-level” programs for z/Architecture processors, not specialized or privileged operating system components. This text therefore deals with nonprivileged instructions, which in any event are the great majority of instructions in all programs.

4. I confess that levels of detail may vary depending on my level of interest in a particular topic.

5. The Exercise and Programming Problem solutions should be considered as samples, and are not in any way intended to be the “correct” solutions.[2] If yours are shorter, simpler, or just plain nicer, so much the better. But if your solutions seem to be two or three times longer than these, you may want to study them for suggestions of workable approaches to solving a programming problem.

6. Some of this material is based on lecture notes I created for Assembler Language classes when I was at the Stanford Linear Accelerator Center in Menlo Park, California.

---

[1] Yes, this book is too long. As my Chinese-restaurant fortune cookie said: “You have a love for words, and should write a book.”

[2] I urge you not to look at them before you're tried your own solutions (or if you're completely stuck at some point). It's OK to learn from someone else's programs, but best it you do it only after you're tried your own.

```
IIIIIIIIII  NN        NN
IIIIIIIIII  NNN       NN
    II      NNNN      NN
    II      NN NN     NN
    II      NN  NN    NN
    II      NN   NN   NN
    II      NN    NN  NN
    II      NN     NN NN
    II      NN      NNNN
    II      NN       NNN
IIIIIIIIII  NN        NN
IIIIIIIIII  NN        NN
```

A digital computer can be considered from various viewpoints; here are five possible views, each treating the computer's inner workings in successively less detail.

- To an engineer concerned with designing its logical circuits, a computer might be thought of as a collection of devices for controlling and ordering the flow of electrical signals.
- At another level, a person concerned with methods used to make these logical circuits perform operations such as addition and division might treat a computer as a collection of registers, switches, and control mechanisms that perform a series of steps leading (say) to the computation of a quotient.
- At the next level one might consider a computer's basic operations to be single arithmetic operations, a simple data movement, or a test of a single piece of data.
- Another viewpoint (typical of “higher-level languages”) considers the basic operations to be moving blocks of data, evaluating and assigning mathematical expressions, and controlling counting and testing operations.
- At yet another level, as in certain applications such as traffic simulation, data reduction, and network analysis, the computer processes information in a form closely approximating the problem under consideration, and produces output directly applicable to that problem.

Each of these views is of course not especially distinct from its neighbors. We will be primarily concerned with the middle level, considering the basic operations or instructions that we want the computer to perform, such as single arithmetic or logical operations, simple data transmission operations, etc. We will also consider the computer from “neighboring” viewpoints: sometimes it is useful to know some details of the internal sequencing of operations such as multiplication and branching; at other times it will be convenient to consider groups of instructions such as macro instructions that perform operations in a larger context.

The level that is our primary concern is usually known as “Assembler Language programming” or “assembler coding”.[3] The assembler we'll describe is the IBM High Level Assembler for z/OS & z/VM & z/VSE, known as “HLASM”. It also can be used on IBM Linux for System z.

Getting the desired machine language instructions and data into the computer in executable form requires the aid of a number of programs: the most important for us is the assembler. Other important programs are the linker[4] and the operating system Supervisor. Each will be considered in the appropriate context.

[3] Some people call it “BAL” — meaning “Basic Assembler Language” — but the language is not basic (nor is it BASIC) except in the sense that it can be fundamental to understanding the System z processor's operations.

[4] The term “linker” here stands for several important programs that combine and load programs for execution. Their names vary among operating systems (Binder or Linkage Editor and Program Loader on z/OS, Loader and Link Editor on z/VM, Linkage Editor on z/VSE, etc.)

To give hardware designers greater freedom to implement instructions in the best way, without your having to be aware of each implementation's techniques, IBM describes an “architecture”. A processor's architecture defines the actions of instructions, I/O, storage, etc. to describe a known set of behaviors, while giving processor designers flexibility in implementing those behaviors.

It will help to have available a copy of the *z/Architecture Principles of Operation* manual. It is easily obtained, and is the reference for basic System z architecture. You should consult it regularly when we discuss individual instructions.

> **Remember!**
>
> The Assembler Language itself is quite simple. The syntax is sparse, there are few “reserved words”, and almost no structuring rules. The main challenge in learning Assembler Language is learning about the processor for which you're writing programs.

Von Neumann Architecture

The IBM System z processor is one of a large class of computers known as “Von Neumann Architecture”, named after John Von Neumann, a mathematician at the Institute for Advanced Study (IAS) in Princeton, NJ, USA. He and colleagues designed a processor in which programs and data shared the same memory. A machine was built to that design in the early 1950s, and the overall design (sometimes called “Institute machines”) was widely adapted in the U.S., Europe, Japan, and Australia.[5]

Why Program in Assembler Language (and Why Not)?

Before going any further, ask why you're considering writing programs in Assembler Language. These are some reasons *for* programming in Assembler Language:

1. You have to.

   Maybe you're taking a course like “Assembler Language Programming”, or you've been made responsible for an existing Assembler Language application.

2. You want to.

   It's useful to know, or maybe you're just curious about what is *really* going on inside the processor when you write high-level language programs. The architecture represented by System/360 and its modern descendants has pervaded the computing industry since the mid-1960's, and will continue to do so for many years. Because you may encounter some modern incarnation of the System/360 family, it helps to be familiar with its architecture.

3. It's educational.

   Programming in Assembler Language is the best way to learn how the processor works. Even if you program in high-level languages, there will be times when understanding the processor's properties will help you understand why certain choices and tradeoffs are made in programming in those languages.

4. It's fundamental.

   A key to writing efficient software is understanding the underlying hardware; no language other than Assembler Language provides such insights. Even if you don't write much Assembler Language code, writing good high-level language programs often depends on knowing how to write good Assembler Language programs.

   Debugging a problem in high-level language applications may require knowing some machine language. (You might say that a language needing this kind of debugging isn't very “high-level”, but it *is* necessary at times.)

[5] Why do I care? My first computer was the ILLIAC I, built to the IAS design.

Assembler Language is also a natural vehicle for recovering lost source code (yes, it happens!). Object or binary programs can easily be disassembled into Assembler Language source programs.

5. It can be more efficient.

   Efficiency depends on many things. Because you can specify almost the exact instruction sequences you want, you can do many things to improve program efficiency. If you know which parts of a program consume the most time, recoding those parts in Assembler Language can often lead to savings.

   However, pursuing efficiency has limits. Programmers have been known to struggle happily over a program modification that will save a few seconds of processor execution time over the program's lifetime.[6]

   There is another objection to using Assembler Language to attain efficiency: some modern compilers can produce quite efficient code for certain applications.[7] However, even clever coding and powerful compilers can't help a badly implemented algorithm. Also, you may have difficulty learning the costs of various high-level language statements.

6. It's independent.

   Error recovery (and avoidance) can be simpler in Assembler Language than with high-level languages.

   You need not rely on the presence of any run-time environment other than the operating system environment in which your program will execute. You can access many services that may not be available to high-level languages.

7. It's more flexible.

   There are some processor instructions and facilities for which higher-level languages provide limited or no support. And even when these facilities *are* supported, their expression in such languages may be inefficient, restricted, or difficult to use. Assembler Language may be the simplest, or even the only, way to access those facilities.

   Unlike many high-level languages, Assembler Language imposes no assumptions about how you *should* (or *must*) structure your programs. Someone else's program structures or concepts of proper programming technique aren't forced on you by the language, and you have more freedom to choose solutions you like.

8. It's more powerful.

   In addition to Assembler Language's efficiency and flexibility, you also have available to you the *entire* repertoire of the processor's instruction set. New instructions on your CPU are usable immediately; you don't need to wait for high-level language compilers to "catch up" to the latest architecture. (Some instructions, though, may require special privileges such as executing in supervisor state.)

9. It's more fun.

   You can do things your own way. You can define the meanings of each and every piece of your program, and not have to be satisfied with assurances that "the compiler (or the system) takes care of that for you".

10. It's controllable.

    Unlike "higher-level" languages, the assembler creates machine language instructions and data in exactly the form and order you specify. It doesn't try to organize (or re-organize) anything for you; there are no "helpful" intermediaries between you and the processor. In a nutshell, "What you write is what you get".

---

[6] And possibly wasting many more seconds of processor time re-assembling and re-linking the program than will be ever saved during its execution! (Yes, I've done that...)

[7] Compilers *do* have occasional errors; finding problems with the generated code is easier if you know Assembler Language.

11. It's stable.

    You needn't worry about re-translating and re-testing programs with new releases of compilers or run-time libraries; the object code won't change each time you re-assemble.

12. It's parameterizable.

    Because assembler languages have been with us for almost as long as computers, a lot has been learned about minimizing the pain of modification: we will see that the Assembler Language is very rich in possibilities for parameterization. That is, you can revise a value in just one place in your program, and the assembler automatically adjusts the portions of your program that depend on that value.

13. It's extensible.

    This is one of the best reasons for programming in Assembler Language: you can define *macro instructions* that have whatever meaning *you* want them to have. You can design and create an entire programming language of your own, and then build other languages on top of that, for as many levels as you like or need. Macro-instructions also provide some “insulation” between your program and the habits of the Operating System under whose control it will run.

    Macro instructions are definitely a highly satisfying aspect of Assembler Language programming. Unfortunately, we don't have room here to describe conditional assembly and macros.

Conversely, there are also reasons for *not* programming in Assembler Language.

1. The language can be verbose.

   Economy of expression is *not* a characteristic of most Assembler Languages *except* (and this is an important exception) for the availability of macro instructions. Usually, you must write more lines of code to do a simple task than if you had chosen a higher-level language.

   This is due mainly to the richness of the z/Architecture processor's instruction set, because the Assembler Language itself is quite simple.

2. The language is very flexible.

   It can be *too* flexible for some users. There are many acceptable ways to use Assembler Language to solve a given problem, and almost all problems can be solved with a small and manageable set of instructions.

3. The language is idiosyncratic.

   To a large extent, the occasional lapses of regularity and coherence in the syntax and semantics of the Assembler Language are due to irregularities in the System z instruction set and architecture: instructions that do similar things may have different syntaxes. Thus, the Assembler Language contains occasional “special cases” and “exceptions to the rules”. (This is of course not unique to Assembler Language!)

4. The language's flexibility means it's easier to make errors.

   While this reason is implicit in the previous three, it also is part of the price you pay for being able to specify everything yourself; you have more chances to make mistakes. We will see that there are good ways to avoid some of the pitfalls that this extra freedom provides.

5. Programs can be harder to debug.

   In some cases, programmers may not write programs so they can detect processing errors, or terminate gracefully. Because programs can be written with great freedom, they might not be organized so that errors do a minimum amount of damage. Similarly, some programmers are often reluctant to insert the extra instructions necessary to leave an easily-followed diagnostic trail for the person (you?) trying to discover why your program did something unexpected.

6. Programs may seem hard to maintain.

   Maintenance costs are much more strongly influenced by the structure and clarity of the code than by the language used to write it. Extensive research has shown little difference in maintenance costs between Assembler Language and high-level languages.

7. Lack of a run-time library.

   Assembler Language programs can be written as a component of a high-level language application. But “stand-alone” Assembler Language applications may not have access to the run-time libraries provided with most high-level languages. By using careful modular design techniques, this lack can be overcome with a set of routines or macro instructions that provide functions shareable among many applications.

   Assembler Language programs can access run-time libraries, so long as they adhere to appropriate programming conventions; this can often reduce programming effort.

8. Lack of portability.

   Unlike programs written in some high-level languages, assembler language code is intended for execution only on the processors for which it is created. (And, high-level language programs are not always easily moved to other processor architectures!)

If none of the reasons *for* programming in Assembler Language has much appeal to you — you don't *have* to program in Assembler Language, you don't need its efficiency, flexibility, power, or extensibility, and your sources of amusement (or employment) lie elsewhere — then *don't*. Use whatever tool will do the job with the least time, effort, and nuisance, and get on to whatever task comes next.

Assembler Language Misconceptions

These are some common misconceptions about Assembler Language:

- Assembler Language is dead.

  For many small-environment or short-lived applications, fast implementation is more important than long life, small size or high performance. But many substantial organizations have made major investments in Assembler Language applications that must be fast, compact, and can process high volumes of data efficiently; such applications need regular enhancement.

- It's hard.

  The language itself is trivially simple. Understanding programs in any programming language is more a matter of clear coding and good style and organization. Any programming task can be made easy or difficult. (We'll offer occasional bits of advice on ways to simplify your programming challenges.)

- Assembler Language programs are faster than compiled programs.

  That depends more on your choice of algorithms, the high-level language, and its compiler. You can write slow programs in any language.

- Assembler Language programs are hard to read.

  Only if you write them that way! But you do need to understand the System z instructions.

- It's hard to manage all those base registers.

  Not at all: careful program organization and appropriate instruction choice easily make it a non-problem.

- Assembler Language is hard to maintain, especially if you don't have the needed skills.

  Extensive research shows little difference in maintenance costs among programming languages, and lack of skills is a problem for any programming language.

- Many applications written in Assembler Language can be replaced with “out-of-the-box” functionality.

  It's rare that a purchased software package does exactly what your organization needs; you must pay in time and money for negotiating, training, and adaptations that you could complete “at home” more promptly and cheaply.

- You don't need to worry about efficiency, because a faster CPU will be along in a year.

Rarely true, because growing businesses continually need to process more data and their programs must provide new capabilities. Once you fall behind, it's difficult to rewrite inefficient applications.

- Converting an Assembler Language application to a high-level language will make it easier to hire skilled programmers.

  There are some critical factors here: research has shown that

  1. Changing language has many hidden costs and should be avoided.
  2. High-level languages do *not* improve reliability or maintainability.
  3. Problem-domain expertise is often more important than programming-language expertise. It it much easier to train people who understand your business and business processes in the language you need, rather than hiring people who have the necessary language skill.

  Some further considerations include:

  - Transition and testing require system stability, which implies possible lost business opportunity.
  - Converting and validating test cases can be a major effort in itself.
  - High stress levels for "bilingual" staff.

- You can't do "Structured Programming" in Assembler Language.
  - By using a set of Structured Programming macros such as those available with the High Level Assembler, programs can be as fully structured as any high-level language program.
  - Because you have full control over the separation of code and data into individual modules, you can have *greater* flexibility in determining the structure of an application than a typical high-level language may provide.

> **Remember!**
>
> **What's good about Assembler Language?**
> Almost everything you do works OK.
>
> **What's bad about Assembler Language?**
> Almost everything you do works OK.

Exercises

0.1.1.(1)+ Why are you interested in Assembler Language?

0.1.2.(0) What is the difficulty level of this exercise?

Chapter I: Getting Started

```
IIIIIIIIII
IIIIIIIIII
    II
    II
    II
    II
    II
    II
    II
    II
IIIIIIIIII
IIIIIIIIII
```

In this chapter, we will look at factors involved in Assembler Language programming, and then investigate the binary number representation and its arithmetic.

- Section 1 looks at some notation, terminology, and conventions we'll use.
- Section 2 describes basic topics about the number representations used in System z processors: binary and hexadecimal numbers, arithmetic and logical representations, 2's complement arithmetic, and discusses alternative number representations.

1. Some Basic Items

```
    11
   111
  1111
    11
    11
    11
    11
    11
    11
    11
1111111111
1111111111
```

In this section we introduce some basic terms and notations that we'll use later, and then investigate the important properties of the binary number representation.

1.1. Notation and Terminology

- Some diagrams and figures need to show the lengths and positions of parts of the figure. In Figure 1 we want to show some object's structure, and to indicate the amount of space required for each of its components. To do this, we place a number *above* the field to indicate its length. In cases where we also indicate the numbering and positions of the digits in that component, we use numbers *below* the field, at its right and left ends. Two four-digit fields in an eight-digit area would be as shown in Figure 1:

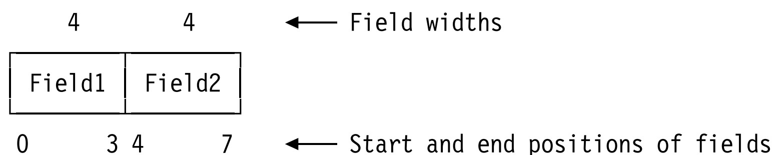


Figure 1. Example of numbering and notation

By convention, numbering starts with digit zero on the left. We call the leftmost digit, digits, or portion of a field the *high-order* part of the field; the rightmost digit, digits, or portion of the field the *low-order* part. Thus, position 0 in this figure is the high-order digit, and position 7 is the low-order digit.

- Standard mathematical symbols such as subscripts and superscripts, and the capital Sigma used to denote summation are hard to produce, so we sometimes use a slightly different notation. For subscripted quantities like B_k ("B-sub-k") we will sometimes use "B_k", but also either "Bk" or the programming-language convention "B(k)". For quantities like "element i,j of ARRAY" or "ARRAY-sub-i,j" (often written "$ARRAY_{i,j}$") we write "ARRAY(i,j)". There are very few places where the juxtaposition of two letters like "XY" means multiplying X and Y, but these will be obvious from the context where they appear. In some cases we use superscripts for quantities like 10^5 and B^k, but we also use the common notation of paired asterisks to denote exponentiation, as in `10**5` and `B**k`.

- For the operations of addition, subtraction, multiplication, and division, we use the operators +, −, *, and / respectively. (In some descriptions, we use "×" for multiplication and "÷" for division.) We use vertical bars or the functional notation ABS() to denote absolute values: |x| and ABS(x) mean the magnitude of the quantity x.
- To denote the contents of something called "x", we use c(x) or C(x). Sometimes the object whose contents we're interested in will be an identifiable object such as a register, so that we might speak of the contents of Register 1 as c(R1). At other times we may speak of the contents of something whose actual form or location is not precisely known, such as an area of memory that has been given the name AREA; in this case we still use the notation c(AREA).
- Some words have similar but different meanings. For example, the word "operand" is used in several different senses in most of the literature describing System z and its Assembler Language.
  1. In the description of instructions in the *z/Architecture Principles of Operation*, an operand is the object being operated on, or is involved in the instruction. For example, in

     $LM \quad R_1,R_3,S_2$

     the contents of general register designated by R_1 is the first operand, and the contents of storage addressed by S_2 is the second operand. Note that operand numbers may not correspond to their sequential position!
  2. In an Assembler Language *statement*, an operand is defined by its position in the operand field. For example, in

     ```
     LM    2,12,SAVE
     ```

     the first operand is 2, the second operand is 12, and the third operand is the symbol SAVE. Note the difference in operand numbering compared to the *z/Architecture Principles of Operation* description!
  3. During execution, an operand is the subject of an operation: an operand is something being acted on or operated on by an *instruction* as it is executed in the processor. For example, in

     ```
     LM    2,12,SAVE
     ```

     one operand is the contents of general register 2, and another operand is the contents of memory named by the symbol SAVE.

  We'll try to clarify these differences; the intended sense will be usually clear from the context in which the word appears.
- Sometimes we need to indicate positions where a blank or space character should appear. Rather than use a blank " " character, we sometimes use a "•" character. For example,

  John•Q.•Public

  has a blank on each side of the middle initial.
- Sometimes we refer to the "euro" character. Because this document formatter doesn't have that exact character, we use ∈ as the best available approximation.

Exercises

1.1.1.(1) What is the total width of the fields illustrated in Figure 1 on page 12?

1.2. Instruction Elements

We will often refer to parts of a machine instruction. In the *z/Architecture Principles of Operation*, you will see notations like

R_1,R_2 or $D_2(X_2,B_2)$ or M_1 or I_2 or L_1

where the subscripted letters specify numeric values appearing in the fields of a machine instruction. They are simply a way to indicate numbers that you or the assembler must provide. In particular:

- A notation like "R_1" is simply a number that usually denotes *any* register, not "register number 1".

- “GR R_1” means the general register denoted by the number used in place of R_1.
- “GR1” means general register 1.

We'll clarify these and other details as we proceed.

1.2.1. Register Names

We refer regularly to registers, using small numbers like 0, 12, etc. Some people like to use “names” like R0 and R12 for them. They can be helpful, but can also be very misleading, because “R0” isn't really a register name; it's only a name for a number. (Some exercises will help you understand why it can be misleading.) I don't want you to develop a habit of thinking that names like “R0” always mean “register 0”. I prefer to use just numbers like “0” or “12” to designate register names.[8] That said, we will sometimes refer to a specific register using terms like R0 and R12, meaning specifically general registers 0 and 12.

Unlike some other processors and their assemblers, there are no reserved register names or symbols in the System z Assembler Language.

Exercises

1.2.1.(1) If R_1 has value 9, what register is referenced by GR R_1?

Terms and Definitions

algorithm
: A finite sequence of well-defined steps for solving a problem. After *al Khwarizmi*, a nickname of the 9th century Persian astronomer and mathematician Abu Jafar Muhammad ibn Musa, who authored many books on arithmetic and algebra. He worked in Baghdad and his nickname alludes to his place of origin, Khwarizm (Khiva), in present-day Uzbekistan and Turkmenistan.

architecture
: A description of “the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, and the physical implementation.”[9]

assembler
: A program that translates programs written in Assembler Language to machine language instructions and data.

Assembler Language
: A lower-level language allowing programmers maximum freedom in specifying processor instructions, providing powerful “macro instruction” facilities supporting encapsulation and economy of expression.

blank
: A nonempty, finite-width invisible character; a space. In contexts where explicit blank spaces appear, we sometimes use the “•” character.

HLASM
: IBM's High Level Assembler for z/OS and z/VM and z/VSE. The assembler we describe here.

---

8 One reason for using symbolic register names was that all early assemblers' “Symbol Cross Reference” (a list of all symbols used in your program) showed the places where the names were used — and searching the cross-reference might be the only way to know which instructions might have referenced specific registers. The IBM High Level Assembler for z/OS & z/VM & z/VSE provides a “Register Cross Reference” showing where the general registers were used, *whether or not they were named*. So, it's no longer necessary to “name” registers.

9 G.M. Amdahl, G.A. Blaauw, and F.P. Brooks, Jr. *Architecture of the IBM System/360*, IBM Journal of Research and Development Vol. 8 No. 2, 1964, reprinted in IBM Journal of Research and Development Vol. 44 No. 1/2, January/March 2000.

operator
A character specifying a mathematical operation: + for addition, − for subtraction, * or × for multiplication, and / or ÷ for division.

space
A nonempty, finite-width invisible character; a blank character. In contexts where explicit blank spaces appear, we sometimes use the "•" character.

2. Binary and Hexadecimal Numbers

```
 2222222222
222222222222
22        22
          22
          22
         22
       22
     22
   22
 22
222222222222
222222222222
```

In this section we examine number representations and methods for converting numbers in those representations to and from decimal. Then we examine arithmetic using numbers in the binary representation.

System z, like most other digital computers, uses binary—base two—numbers for most internal arithmetic. A binary digit takes only values 0 and 1; because it is relatively simple to build a mechanical or electrical device representing a binary digit, the binary representation is quite natural. For example, a 1 digit may be represented by the presence of a current through a circuit component or by the presence of a positive voltage at some point. Facility with binary numbers is fundamental to understanding the basic operations of System z, so it is important to understand the binary number representation.

For now, all numbers are assumed to be integers. This means that the “decimal point” (the “radix point” or “binary point”) lies at the right end of the number. We will discuss nonintegral (fractional) numbers in Sections 29 and 31.

We are familiar with numbers using radixes other than 10. Times (and angles) measure minutes and seconds using radix 60; hours are counted using radix 24; and before The United Kingdom changed to a decimal monetary system: radix 20 for shillings and radix 12 for pence. Binary is easier.

2.1. Positional Notation and Binary Numbers

In base ten, writing a number such as “1705” means the quantity

$1000 + 700 + 00 + 5,$

which can also be written as

$1\times1000 + 7\times100 + 0\times10 + 5\times1,$

or as

$1\times10^3 + 7\times10^2 + 0\times10^1 + 5\times10^0.$

That is, each digit position as we move to the left is weighted by one more power of the base, ten.

Similarly, when in binary notation we write "11010" we mean

$$10000 + 1000 + 000 + 10 + 0,$$

$$1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0,$$

$$1\times16 + 1\times8 + 0\times4 + 1\times2 + 0\times1$$

which is *not* the same as what is meant by the *decimal* number 11010, where powers of ten are understood. In fact, the *binary* number 11010 is the representation (in the number system with base two) of the decimal number 26: the sum in this example is 16+8+2.

To clarify which base is intended we use a notation like the Assembler's: if base 10 is intended, the digits are written normally; if base 2 is intended, the binary digits are preceded by a "B" and an apostrophe, and are followed by an apostrophe. For example:

```
26 = B'11010', 1 = B'1', 10 = B'1010', 8 = B'1000', 999 = B'1111100111'.
```

Positional notation can be used for any base (or *radix*). For example, if humans had only one hand we might use base 5 for numbering, so that 1413 in base 5 would have decimal value 233 (in our ten-finger decimal world):

$$\begin{aligned} 1413_5 &= 1\times5^3 + 4\times5^2 + 1\times5^1 + 3\times5^0 \\ &= 125 + 100 + 5 + 3 \\ &= 233_{10} \end{aligned}$$

Exercises

2.1.1.(1)+ Determine the decimal value of the following binary numbers: (a) B'000010110', (b) B'000101100', (c) B'10101010', (d) B'1111111'.

2.1.2.(1)+ Suppose a binary number is represented by a single 1-bit followed by a string of *n* zero bits (100...00). What is its value?

2.1.3.(2) Suppose a binary number is represented by a string of *n* one bits (111...11). What is its value?

2.2. Hexadecimal Numbers

As values become larger, the number of binary digits required becomes larger also (over three times as many bits as decimal digits), so we use a more compact notation for binary numbers. If we consider groups of four binary digits at a time, the possible decimal values that can be represented run from zero to fifteen. If we then represent each of these groups by the "digits" 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, we can establish the correspondences shown in Table 1 on page 18. (The letters A through F are a natural choice for "digits", but we could actually have chosen any other six symbols to represent the "digits" to which we assign the values 10, 11, ..., 15.[10])

We use the same positional notation for base 16 number representation as for decimal and binary numbers. Thus, we can write the base 16 number $A97E_{16}$ as

$$A\times16^3 + 9\times16^2 + 7\times16^1 + E\times16^0,$$

or

$$10\times16^3 + 9\times16^2 + 7\times16^1 + 14\times16^0 = 10\times4096 + 9\times256 + 7\times16 + 14 = 43390.$$

[10] In fact, some early computers such as the ILLIAC I used the characters K, S, N, J, F, and L because those letters had the required binary 4-bit hole combinations on 5-hole punched paper teletype tape. (Remembering those six letters was helped by the phrase "Kind Souls Never Josh Fat Ladies".)

Why use something as unfamiliar as a base-sixteen representation for numbers that are binary in nature? Base 16 is compact and convenient for expressing long strings of binary digits, and a natural representation for System z. Other groupings are possible; another form is “octal”, or base eight, in which the binary digits are grouped by threes.[11]

Table 1. Binary, decimal, and hexadecimal

Binary Digits	Decimal Value	Hex Digit
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

The base sixteen digits in the third column are called *hexadecimal*[12] or *hex* digits, and we use them in most situations when we need to refer to binary numbers. As with binary numbers, a notation similar to the Assembler's will denote hexadecimal quantities: the hexadecimal digits are preceded by an “X” and an apostrophe, and are followed by an apostrophe. For example:

```
26 = B'11010' = X'1A',     X'26' = B'100110' = 38,

1 = B'1' = X'1',           10 = B'1010' = X'A',

B'1000' = 8 = X'8',        100 = X'64' = B'1100100'.
```

Converting numbers between binary and hexadecimal representations is easy:

- To convert a hexadecimal number to binary, substitute for each hexadecimal digit the four binary digits it represents.
- To convert a binary number to hexadecimal, group the binary digits four at a time starting from the right (adding extra zeros at the left end if needed), and substitute the corresponding hexadecimal digit.

For example:

```
X'D5B' = B'1101 0101 1011'       (hexadecimal to binary),

B'11 1110 1000' = X'3E8'         (binary to hexadecimal).
```

In the second example we could add two extra binary zero digits at the left or “high-order” end of the number without affecting its value; similarly, we can omit high-order zero digits, and write

```
X'11' = B'10001'   (rather than B'00010001').
```

[11] Processors whose word lengths were “natural” multiples of 3 included the IBM 70x and 709x processors with 36-bit words, and several Control Data Corporation (CDC) processors with 48-bit words. Most processors now have word lengths that are a multiple of 8 bits.

[12] The correct term for base 16 is “sexadecimal” (or even “hexadecadic”), but you can understand that abbreviating the term “sexadecimal” would not be appropriate for dignified corporations.

Don't omit zeros on the right! That is, B'00111100' ≠ X'F'.

Converting between decimal and hexadecimal representations is more cumbersome; it is simplest to use Tables 2 and 3 starting on page 20 below, and the tables in "Appendix A: Conversion and Reference Tables" on page 995. The following section discusses general methods for converting integers from one base to another; if you are satisfied to use the tables, the next section may be skipped.

We use these abbreviations regularly: *bit* means "binary digit", and *hex* is an abbreviation for "hexadecimal".

Exercises

2.2.1.(1) Convert the following hexadecimal numbers to binary: (a) X'A', (b) X'2B', (c) X'3E8'.

2.2.2.(1) Make a table similar to Table 1 on page 18 showing binary, decimal, and octal (base 8) values.

2.2.3.(2) In grouping bits to form hex digits, why can't we start at the left? That is, why do we begin grouping at the radix point?

2.2.4.(2)+ Create addition and multiplication tables for single hexadecimal digits.

2.2.5.(1) Convert the following octal numbers to hexadecimal:

1. 21474
2. 77777
3. 1750
4. 60341303
5. 4631

2.2.6.(3) You may have noticed that the characters in many cartoons and comics have only four fingers. To help them with "cartoon arithmetic", create base-8 (octal) addition and multiplication tables.

2.3. Converting Integers from One Base to Another (*)

In our familiar notation, a string of digits like 73294 in some base A means

$$7\times A^4 + 3\times A^3 + 2\times A^2 + 9\times A^1 + 4\times A^0.$$

Using symbols, the digit string

$$d_n \; \ldots \; d_3 \; d_2 \; d_1 \; d_0$$

is the representation in some base A of a number X:

$$X = d_n\times A^n + \ldots + d_3\times A^3 + d_2\times A^2 + d_1\times A^1 + d_0\times A^0.$$

The subscripts on the digits d match the power of the base A. If A has value 10, then the digit string 73294 is the familiar decimal number seventy-three thousand, two hundred ninety four.

Suppose we want to convert X from its representation in base A to its representation in a new base B, with digits e_0, e_1, e_2, etc.:

$$X = e_m\times B^m + \ldots + e_3\times B^3 + e_2\times B^2 + e_1\times B^1 + e_0\times B^0.$$

We know the old and new bases A and B, and the digits d_k of the old representation. To find the digits e_k of the new representation, we use the following scheme;

1. Divide X (in base A notation and arithmetic) by the new base B; save the quotient. The remainder is the low-order digit e_0. This can be seen from the definition of the quotient and remainder:

```
X = B × Quotient + Remainder
  = B × [e_m×B^(m-1) + ... + e_3×B^2 + e_2×B^1 + e_1×B^0] + e_0.
```

where the term in square brackets is the quotient.

For example, taking A to be 10 and B to be 16, we convert 73294 to hex:

```
X = 73294 = 16 × Quotient + Remainder = 16 × 4580 + 14,
```

so e_0 = 14 = X'E'.

2. Now, divide the saved quotient by B; save the new quotient, and the new remainder is e_1.

   In our example, dividing 4580 by 16 gives quotient 286 and remainder 4, the value of the next digit, e_1.

3. Continue this process until a zero quotient is obtained. The successive remainders are the desired digits e_0, e_1, ..., e_m; they were obtained in order of *increasing* significance, from *right* to *left*.

   Continuing to divide by 16 in our example, we obtain remainders 14, 1, and 1; these are the digits e_2, e_3, and e_4 respectively. The result of this sample conversion shows that 73294 (base 10) has value 11E4E (base 16).

Our most frequent conversions are between decimal and binary or hexadecimal; use Tables 2 and 3, or the conversion tables in Appendix A.

1. If the number is small enough, find it in the conversion tables.
2. For larger numbers,
   a. To convert from hex to decimal, find each digit's decimal value in the tables in Tables 2 and 3, and evaluate the sum.
   b. To convert from decimal to hex, find the largest power of 16 in the tables that is less than or equal to your number, subtract that number, and note the corresponding hex digit. Repeat, writing the hex digits from left to right. The following example shows how to do this for the decimal value 1000:

```
1000
-768   hex digit 3
 232
-224   hex digit E
   8
  -8   hex digit 8
   0
```

   so that 1000 (decimal) is X'3E8'.

Table 2. Multiples of powers of sixteen (part 1 of 2)

Hex Digit	**× 16^0**	**× 16^1**	**× 16^2**	**× 16^3**	**× 16^4**
1	1	16	256	4,096	65,536
2	2	32	512	8,192	131,072
3	3	48	768	12,288	196,608
4	4	64	1,024	16,384	262,144
5	5	80	1,280	20,480	327,680
6	6	96	1,536	24,576	393,216
7	7	112	1,792	28,672	458,752
8	8	128	2,048	32,768	524,288
9	9	144	2,304	36,864	589,824
A	10	160	2,560	40,960	655,360
B	11	176	2,816	45,056	720,896
C	12	192	3,072	49,152	786,432
D	13	208	3,328	53,248	851,968
E	14	224	3,584	57,344	917,504
F	15	240	3,840	61,440	983,040

Table 3. Multiples of powers of sixteen (part 2 of 2)			
Hex Digit	× **16**^{5}	× **16**^{6}	× **16**^{7}
1	1,048,576	16,777,216	268,435,456
2	2,097,152	33,554,432	536,870,912
3	3,145,728	50,331,648	805,306,368
4	4,194,304	67,108,864	1,073,741,824
5	5,242,880	83,886,080	1,342,177,280
6	6,291,456	100,663,296	1,610,612,736
7	7,340,032	117,440,512	1,879,048,192
8	8,388,608	134,217,728	2,147,483,648
9	9,437,184	150,994,944	2,415,919,104
A	10,485,760	167,772,160	2,684,354,560
B	11,534,336	184,549,376	2,952,790,016
C	12,582,912	201,326,592	3,221,225,472
D	13,631,488	218,103,808	3,489,660,928
E	14,680,064	234,881,024	3,758,096,384
F	15,728,640	251,658,240	4,026,531,840

The binary powers 2^{10}, 2^{20}, and 2^{30} are often abbreviated by the letters “K”, “M”, and “G”. Thus, it is common to refer to the decimal number 4,096 = 2^{12} as “4K”. Similarly, 3×2^{20} might be referred to as “3M”. Thus, for example, an area of memory (which we'll discuss in Section 3.1) containing 8,192 storage locations might be said to contain “8K bytes” or “8 K-bytes”.[13]

Exercises

2.3.1.(2)+ Convert these numbers from the given base to the new bases.

1. 26293 (base 10) to bases 2, 4, 8, and 16.
2. `X'2FACED'` (base 16) to bases 10 and 2.
3. `X'BABEF00D'` (base 16) to bases 10 and 8.
4. `X'C0FFEE'` (base 16) to bases 10 and 2.

2.3.2.(2) Convert the following to decimal.

1. `X'7FFFFFFF'`
2. `X'C1C2C3'`
3. `X'4040405C'` (This digit pattern will reappear in other forms!)

2.3.3.(3) Make a table of the hexadecimal values of the squares of the integers from 1 to 32.

2.3.4.(2)+ Convert the following hexadecimal numbers to decimal.

1. `X'257'`
2. `X'7FFA'`
3. `X'8008'`
4. `X'E000'`
5. `X'FFFA'`
6. `X'E1010'`

2.3.5.(3) Suppose we must convert a number from its representation in base A to its representation in base B. In which base will it be most convenient to do the arithmetic involved in the conversion? How does the result depend on the base used for the conversion?

2.3.6.(2) Convert these octal (base 8) numbers to base 10: (a) 5061, (b) 257, (c) 192. Work carefully!

---

[13] More properly, the abbreviations K, M, and G refer to the closest powers of 10: one thousand = 1K = 10^3, one million = 1M = 10^6, etc. To avoid this confusion, you can use the more precise terms “Ki”, “Mi”, and “Gi” to refer to the binary powers. But few computer people bother.

2.3.7.(2) What decimal values are represented by the binary numbers 9K, 5M, and 2G?

2.4. Examples of General Conversions (*)

We will use the division methods described in the previous section to illustrate conversions from one base to another.

1. Convert 19 (base 10) to base 2.

```
   9        4        2        1        0
2)19     2)9      2)4      2)2      2)1
  18       8        4        2        0
   1=e0     1=e1     0=e2     0=e3     1=e4
```

   Hence, 19 = B'10011'.

2. Convert 1000 (base 10) to base 16. (The conversion arithmetic is done in base 10.)

```
    62            3               0
16)1000       16)62           16)3
   992           48               0
     8=e0        14 (X'E')=e1     3=e2
```

   Hence 1000 = X'3E8'.

3. Convert 627 (base 10) to base 9.

```
   69       7        0
9)627    9)69     9)7
  621      63       0
    6=e0     6=e1     7=e2
```

   so that 627 (base 10) = 766 (base 9).

4. Convert 766 (base 9) to base 7. First, we convert to base 10, and then do the arithmetic in decimal:

   766 (base 9) = 7×81 + 6×9 + 6 = 567 + 54 + 6 = 627 (base 10)

```
   89       12       1        0
7)627    7)89     7)12     7)1
  623      84        7        0
    4=e0     5=e1     5=e2     1=e3
```

   so that 766 (base 9) = 1554 (base 7).

   If you are mathematically inclined:

   Just for fun, now do the conversion in base 9:

```
  108       13        1        0
7)766    7)108     7)13     7)1
  762      103        7        0
    4=e0     5=e1      5=e2     1=e3
```

   Thus 766 (base 9) = 1554 (base 7) again. This shows that you can do base conversion using any (other) base for the arithmetic.

5. Convert 1413 (base 5) to base 10. This is simplest if we expand the positional notation:

   1413 (base 5) = 1×125 + 4×25 + 1×5 + 3 = 233_{10}.

If you are still (or very) mathematically inclined:

Alternatively, since 10 (base 10) = 20 (base 5), we can do the conversion in base 5 arithmetic:

```
       43               2              0
  20)1413          20)43          20)2
     130              40              0
      113              3=e1           2=e2
      110
        3=e0
```

Again, we find 1413 (base 5) = 233 (base 10).

6. Convert X'3E8' to base 10. In this case it is simpler to evaluate the positional notation:

   X'3E8' = $3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0$,

   and then evaluate this sum in decimal. Thus we find

   X'3E8' = $3 \times 256 + 14 \times 16 + 8 = 768 + 224 + 8 = 1000$.

This type of conversion can be simpler if you use the table of multiples of powers of 16 in Tables 2 and 3, or the conversion tables in Appendix A.

Exercises

2.4.1.(2) Perform the indicated conversions. For number bases greater than 10, assume that the "digits" corresponding to 10, 11, 12, etc., are represented by the letters A, B, C, etc., respectively.

1. Convert 31659 (base 10) to bases 8, 4, and 2.
2. Convert 6917 (base 10) to bases 5, 13, and 16.
3. Convert X'EF2A' (base 16) to bases 10 and 13.

2.4.2.(2)+ Make a table of the hexadecimal representations of the first ten powers of ten, from 10^0 to 10^9. (Suggestion: use hexadecimal arithmetic, and multiply each term by X'A' to obtain the next.)

2.4.3.(3) Make a table like those in Tables 2 and 3, except that the nine multiples of the powers of ten from 0 to 9 should be expressed in hexadecimal notation.

2.4.4.(3) Convert B'1111101000' to base 10 using binary arithmetic (that is, divide by B'1010').

2.4.5.(3) Convert 73294 (base 10) to bases 11, 12, 13, 14, and 15. Can you make any use of the result of converting to base N to help in converting to base N+1?

2.4.6.(3) Make a base seven multiplication table. Use it to perform the following conversions *directly,* without first converting to base ten: (1) 526 (base 7) to base 16, (2) 10110 (base 7) to base 8, (3) 61436 (base 7) to base 8, (4) 666 (base 7) to base 10.

2.4.7.(2) Convert 629 (base 11) to bases 10 and 12.

2.4.8.(3) In converting from some base A to base 10, it is usually most convenient to expand the positional notation as illustrated in Examples 5 and 6 of Section 2.4. We can also expand the positional form by rewriting it in "nested" form:

$$X = (((\ldots(d_n \times A)+\ldots+d_3) \times A+d_2) \times A+d_1) \times A+d_0.$$

That is, the leftmost digit is multiplied by A, the next digit is added to it and the result is multiplied by A, and so forth until the rightmost digit has been added. Using this technique, perform the following conversions.

1. 2F3 (base 25) to base 10.
2. 61436 (base 8) to base 10.

3. X'DEFACE' (base 16) to base 10.
4. 999 (base 10) to base 16.

2.4.9.(2) In applying the "nested multiplication" technique of the previous exercise to conversions from base A to base B, what base should be used for the conversion arithmetic?

2.4.10.(3) Using the base seven multiplication table you made in Exercise 2.4.6, perform the following conversions in base 7 arithmetic: (1) 526 (base 10) to base 16, (2) 10110 (base 2) to base 5, (3) 61436 (base 8) to base 10, (4) 666 (base 10) to base 7.

2.4.11.(3) Write the decimal value 8 in bases 8, 7, 6, 5, 4, 3, 2, and 1.

2.4.12.(3) If you have two numbers in bases A and B, what is a necessary relationship between A and B that will allow you to use the same "digit grouping" technique you used to convert between binary and hexadecimal?

2.4.13.(3) Show the value of 16_{10} in bases 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, and 3.

2.4.14.(4) Using base 7 arithmetic, calculate the sum and product of 435_7 and 64_7. First, convert those two numbers to base 10 and then add and multiply the results in base 10; then use those results to test that you have evaluated the base 7 sum and product correctly.

2.4.15.(2) Convert the following decimal values to base 3: 2, 6, 10, 12, 16, 28, 41, 99, 104.

2.5. Number Representations

Now that we know how to convert numbers between binary and hexadecimal, we will see how they are used in System z for address calculations, indexing, and integer arithmetic. Up to now, we have examined the binary number representation only for nonnegative numbers; representing negative numbers requires further consideration.

There are three fixed-point (integer) number representations in common use: the *radix-complement,* the *sign-magnitude,* and the *diminished radix-complement* representations. In practice, the widely-used radix-complement representation is called the *two's complement* representation, and the diminished radix-complement representation is called the *ones' complement* representation.[14] Two of these representations are used in System z: the two's complement form is used for addressing and integer arithmetic, and the sign-magnitude form is used for floating-point and packed decimal numbers. A variation of the radix-complement form is used internally for packed decimal arithmetic, which we'll see in Chapter VIII.

With so many representations, you might wonder why the System z designers settled on two's complement. The reason follows from the processor's "architecture": since virtually all computers use the two's complement representation for *address* arithmetic, and because in System z the general registers are used for both arithmetic and addressing, it is natural that ordinary integer arithmetic has the same form.

We will illustrate the following discussion using 32-bit numbers, corresponding (as we shall see) to the length of a word in memory and half the length of a general register.[15]

---

[14] Why is it called "two's complement"? The name of the ones' complement representation seems obvious: just complement each bit by subtracting it from 1 (or, change 0 to 1 and 1 to 0); but we don't get the two's complement by subtracting each bit from 2! We'll explain this oddity shortly.

[15] z/Architecture provides 64-bit general registers, but for now our examples will use the 32-bit length.

2.6. Logical (Unsigned) Representation

To begin, we examine what is represented by the rightmost 32 bits of any *nonnegative* integer.

This	is represented by:
0	X'00000000'
1	X'00000001'
130	X'00000082'
$2^{24}-1$	X'00FFFFFF'
$2^{31}-1$	X'7FFFFFFF'
2^{31}	X'80000000'
$2^{32}-1$	X'FFFFFFFF'
$2^{32}+1$	X'100000001'

Thus, if a number is less than 2^{32}, its value can be held correctly in the 32 available bits. If it is greater than or equal to 2^{32}, some significant bits are lost off the left end. (That is, the number's value is represented modulo 2^{32}.) Some instructions perform unsigned addition and subtraction with numbers that satisfy the inequalities

$$0 \le x \le 2^{32}-1.$$

Such arithmetic is called *logical* or *unsigned* arithmetic; we call this the *logical* or *unsigned* representation of binary numbers. If the 32 bits of a logical binary integer are denoted $b_{31},b_{30},...,b_1,b_0$ (this temporary scheme is the reverse of the field-numbering convention introduced in Figure 1 on page 12), then the value X represented by the binary digits $b_{31}b_{30}...b_1b_0$ is

$$X = b_{31}\times2^{31} + b_{30}\times2^{30} + ... + b_2\times2^2 + b_1\times2^1 + b_0\times2^0$$

in the logical representation. This is the most common numeric interpretation of a string of bits.

The representation of a nonnegative 32-bit number less than 2^{31} is the same in the sign-magnitude, ones' complement, and two's complement representations (and is also the same as its logical representation), no matter which of the three forms is chosen to represent negative numbers. Since the two's complement representation is used for most integer arithmetic in System z, we will investigate its properties in detail. Arithmetic using binary numbers in this representation will be covered shortly.

Exercises

2.6.1.(2)+ Give the decimal value of the following hexadecimal numbers in the logical representation:

1. X'DEADBEEF'
2. X'FFFFFFFF'
3. X'DECODED1'

2.7. Two's Complement (Signed) Representation (*)

This section describes the mathematical justification for the two's complement representation. You can skip to Section 2.8 where a simple "recipe" for calculating the two's complement of a number is shown on page 27.

Most programs must deal with both positive and negative numbers. A single bit (usually, the leftmost) is used to represent the number's sign. A 0 bit represents a "+" sign, and a 1 bit represents a "−" sign.

First, the two's complement representation of a 32-bit *nonnegative* binary integer Y satisfying the inequalities

$$0 \le Y \le 2^{31}-1 \quad \text{(numbers within that range with a “+” sign bit)}$$

is the same as the logical representation. $2^{31}-1$ is the largest integer that can be represented using 31 bits; the remaining (32nd) digit at the left end is zero, the sign digit.

Now, consider negative numbers. The two's complement representation of a *negative* integer Y satisfying the inequalities

$-2^{31} \le Y \le -1$ (numbers within that range with a "–" sign bit)

is simply $2^{32}+Y$. The bit pattern representing this value can be found this way. The leftmost bit is set to 1 to indicate that the number is negative, and the remaining 31 bits are set to the binary representation of the nonnegative integer $(2^{31}+Y)$. The result therefore satisfies the inequalities

$0 \le 2^{31}+Y \le 2^{31}-1$.

The reasons for representing negative numbers this way are not obvious, but we will see that it leads to very simple rules for performing arithmetic on signed binary numbers.

In effect, we have done the following: if Y is positive, we find its value by adding the individual terms $(b_i \times 2^i)$; because the leftmost (sign) bit is zero, it does not contribute to the sum. If Y is negative, the sum of the rightmost 31 bits is $(2^{31}+Y)$, and the leftmost bit is 1. Now, if we assign value -2^{31} to the sign bit, we can combine these to obtain

$$Y = (-2^{31}) \times b_{31} + b_{30} \times 2^{30} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0,$$

where the digits b_{30} through b_0 are the representation of $2^{31}+Y$, not the representation of $|Y|$, the absolute value of Y. This formula is almost the same as that used for the logical representation, except that the leftmost bit has negative "weight". (There are good reasons to assign -2^{31} to the sign bit.)

Finally, we will see how the representations of positive and negative numbers work together. The relationship between the logical and two's complement representations is seen by examining the above sum for the logical representation of X:

$$X_{logical} = b_{31} \times 2^{31} + b_{30} \times 2^{30} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0.$$

If b_{31} is zero, the logical and two's complement representations yield the same value, and $Y_{arith} = X_{logical}$. Now, suppose we are given the 32-bit two's complement representation of a negative number Y_{arith}, and we want to know the value those 32 bits would represent if we consider them as the *logical* representation of a number $X_{logical}$. Since bit b_{31} is 1, indicating a negative number, and we represent the remaining 31 bits of Y_{arith} by $(Y_{arith}+2^{31})$, we find that

$$X_{logical} = 2^{31} + (Y_{arith} + 2^{31}) = (Y_{arith} + 2^{32}) \text{ (modulo } 2^{32}).$$

This is interesting: because we can only represent numbers less than 2^{32} in the 32-bit logical representation, $Y_{arith}+2^{32}$ for *nonnegative* Y must have the *same* bit pattern as $X_{logical}$, since the extra (2^{32}) bit is lost. Thus, for

$0 \le X_{logical} \le 2^{32}-1$ and $-2^{31} \le Y_{arith} \le 2^{31}-1$,

we have the following *key relation* between the logical and two's complement representations:

$$X_{logical} = (Y_{arith} + 2^{32}) \text{ (modulo } 2^{32}).$$

That is, *the bit pattern corresponding to the two's complement representation of any positive or negative number* $-2^{31} \le Y \le +2^{31}-1$ *is the rightmost 32 bits of the sum* $2^{32}+Y$ (modulo 2^{32}).

Why it is called "two's" complement?

This equation is the original source of the term "two's complement". In the earliest computers it was customary to treat binary numbers as fractions: the representation was the same as just described, except that the "binary point" or "radix point" was assumed to lie just to the right of the sign bit rather than at the right-hand end of the number, so that values were in the range $-1 \le$ value $< +1$. The equation giving the relationship between logical and arithmetic representations was then written

$X = Y + 2$ (modulo 2),

so that the representation of a negative number was obtained by finding its complement with respect to 2: its "two's" complement.

Calculating the two's complement representation of a negative number is very cumbersome if we follow the above steps for any negative number Y: we would first have to calculate the binary representation of the positive quantity $2^{31}+Y$. But calculating $(2^{31}-1+Y)+1$ instead is very simple, because the representation of $2^{31}-1$ is exactly 31 one-bits. Now, because Y is negative,

$$2^{31}-1 + Y = (2^{31}-1) - |Y|.$$

Thus |Y|, the magnitude of Y, is subtracted from a string of 31 one-bits. But wherever |Y| has a one bit, the resulting difference bit will be zero, and vice versa. Thus, there is no need to subtract! Just change each of the 31 bits of |Y| to its opposite (namely the result of subtracting it from 1), and we have the value of $2^{31}-1-|Y|$. This result is called the "*ones' complement*" of |Y|. Finally, we add 1 to the rightmost bit position to get $2^{31}+Y$, set the leftmost bit (the sign bit) to 1, and we are done.

If Y_{arith} was nonnegative, complementing *all 32* bits automatically sets a 1-bit in the sign; and if Y_{arith} was negative, complementing all 32 bits sets the sign to zero. So, we don't have to do anything special with the sign bit!

This simple method lets us find the binary representation (in the two's complement representation) of a *negative* number, as we will see in the next section.

Exercises

2.7.1.(3) Convert `X'ABODE'` to base 15, using hexadecimal arithmetic throughout.

2.7.2.(3) We saw that the radix-complement representation of a number Y in radix r with n digits is

$$r^n+Y \text{ (modulo } r^n)$$

Suppose r=10 and n=4. Show the ten's-complement representation of the following values, and indicate which are and are not validly representable.
(1) +729, (2) −729, (3) −1, (4) +9999, (5) −5000, (6) +5000, (7) −9999.

2.7.3.(2) What is the decimal value of the 12-bit binary number 100000000001 in a signed two's complement representation?

2.7.4.(3) Based on your results of Exercise 2.7.3, give an expression for the value of the n-bit binary number 10000...000001 in a *signed* two's complement representation.

2.7.5.(3)+ Knowing the logical representation of the three numbers in Exercise 2.6.1, convert them to their signed decimal representation.

2.8. Computing Two's Complements

A simple scheme for computing two's complements is based on the observation that the representation of a negative number Y is simply $2^{32}-|Y|$.

Two's-Complementation Recipe

Given a binary number Y, to find the two's complement representation of −Y:

1. Take the ones' complement of all bits of Y: change 0 digits to 1, and 1 digits to 0.
2. Add 1 in the low-order (rightmost) position, and ignore carries out of the leftmost position.

These two examples do the arithmetic with eight binary digits rather than thirty-two.

1. Find the two's complement representation of −2.

```
(1) representation of +2:    0000 0010
(2) form ones' complement:   1111 1101
(3) add one:                 +       1
                             1111 1110
```

2. Find the two's complement representation of −75.

```
(1) representation of +75:   0100 1011
(2) form ones' complement:   1011 0100
(3) add one:                 +       1
                             1011 0101
```

This recipe also works in the opposite direction.

3. Find the two's complement representation of B'11111110' (–2).

```
(1) form ones' complement:    0000 0001
(2) add one:                  +       1
                              0000 0010
```

This is the binary representation of +2; thus the two's complement of the two's complement of a number is the original number. So, our recipe for computing complements does not depend on the sign of the original operand.

Two unusual cases arise during complementation when all the bits except the sign bit are zero: the complemented result is the same as the original operand.

4. Find the two's complement representation of B'00000000'.

```
(1) form ones' complement:    1111 1111
(2) add one:                  +       1
    (carry one off left end)  0000 0000
```

The result is zero, and the carry of a 1 bit out the left-hand end is lost. Thus the negative of zero is still zero. This is mathematically satisfying: there is no negative zero.[16]

5. Find the 8-bit two's complement representation of B'10000000'.

```
(1) form ones' complement:     0111 1111
(2) add one:                   +       1
                               1000 0000
```

In this case, the complement of the number is also the same as the original number. This particular number, a negative sign bit with all other bits zero, is called the "maximum negative number". It is well defined, and behaves normally in all arithmetic operations except that is has no representable negation.

The maximum negative number has no corresponding positive value available for the representable negative value. We say that we have generated an *overflow* condition—the result is too large to fit into the number of bits allotted for it. Overflow will be treated in more detail in the following sections on two's complement arithmetic.

Some examples of numbers in the 32-bit arithmetic representation are shown in Table 4 on page 29.

[16] Some older computers used the ones' complement representation for binary integers, so negative zeros were possible. System z packed decimal and floating-point numbers (discussed in Chapters VIII and IX) support negative zeros.

Decimal Value	*32-bit Two's Complement Representation*
0	X'00000000'
1	X'00000001'
256	X'00000100'
5000	X'00001388'
+2147483647 $(+2^{31}-1)$	X'7FFFFFFF'
−2147483648 (-2^{31})	X'80000000'
−2147483647 $(-2^{31}+1)$	X'80000001'
−5000	X'FFFFEC78'
−256	X'FFFFFF00'
−2	X'FFFFFFFE'
−1	X'FFFFFFFF'

Table 4. Examples of two's complement representation

The number of values with positive sign is the same as the number of values with negative sign, since every bit may be chosen arbitrarily. Because zero has a positive sign bit, it is sometimes treated as a positive number, even though (mathematically) it has no sign. If we exclude zero as a positive number, then there is one fewer member of the set of positive values than of the set of negative values, since there is no representation for $+2^{31}$. With 32 bits, we can represent 2^{32} values: between -1 and -2^{31} there are 2^{31} values; 0 is a single value; between +1 and $+2^{31}-1$ there are $2^{31}-1$ values. The total number of possible signed values is therefore $2^{31}+1+(2^{31}-1)$, or 2^{32}.

Unfortunately, the terminology used to describe this process can be confusing. We are actually describing the mathematical operation of *negation* that turns a value into its negative. For other number representations, the operation that forms the negative of a number will be different, because there are many ways to represent a negative number. However, sometimes *complementation* is used to describe the operation of negation! For example, we often talk about the binary representation of some number, and then say that in negating that quantity we have formed its *two's complement*.

Exercises

2.8.1.(1) Why does the simple two-step prescription for computing complements given above not depend on the sign of the number being complemented?

2.8.2.(2)+ Give the decimal values represented by each of the following 16-bit numbers, assuming that the binary values are in two's complement representation:

1. X'0257'
2. X'7FFA'
3. X'8008'
4. X'E000'
5. X'FFFA'

(See Exercise 2.3.4. also.)

2.8.3.(2) It is sometimes said that the complement of a number X is the same as −X. State this more precisely.

2.8.4.(2) Four 16-bit areas of a program are named A, B, C, and D. Their contents are

```
c(A) = X'7D40'
c(B) = X'D000'
c(C) = X'15A2'
c(D) = X'800A'
```

If they are the signed 16-bit two's complement binary representations of four decimal numbers, determine their decimal values.

2.8.5.(2) Given the quantities Z = 0, A = 1, B = 9, C = 62, D = 101, E = 255, F = 256, give the nine-bit (eight bits plus sign) representations of the positive *and* negative values of each quantity in the two's complement representation.

2.8.6.(3) Give the 32-bit two's complement representation (in either hexadecimal or binary) of *both* the positive and negative values of the following decimal integers: (1) 10, (2) 729, (3) 1000000, (4) 1000000000, (5) 2147483648, (6) 65535, (7) 2147483647.

2.8.7.(3) Sometimes two's complementation is described by these steps:

- Subtract 1
- Complement all bits

Does this differ from the two's complementation recipe given on page 27? Create examples that show how this form does or does not differ from that recipe.

2.8.8.(1) Give the 16-bit two's complement binary representation of each decimal number in hexadecimal.

1. +13055
2. −9582

2.8.9.(2)+ Show the 32-bit hexadecimal value of the two's complement binary representation of each of the following decimal values.

1. +5
2. −97
3. +65795
4. −16777158
5. +16777219
6. −78606

2.8.10.(1)+ Assuming a 16-bit two's complement representation, give the signed decimal values of these hexadecimal values.

1. `X'B00F'`
2. `X'FFF1'`
3. `X'0FFF'`
4. `X'F001'`

2.9. Sign Extension

In the representation of nonnegative numbers, an arbitrary number of zero bits may be attached to the left end of a number without affecting its value. For example, the 8-bit and 16-bit representations of +9 are

```
B'0000 1001'   and   B'0000 0000 0000 1001'
```

respectively. Similarly for negative numbers, we can add any number of 1 bits at the left without affecting the value. For example, the 8-bit and 16-bit two's complement representations of −9 are

```
B'1111 0111'   and   B'1111 1111 1111 0111'
```

respectively. Thus, for numbers that can be represented correctly in a given number of bits, the correct representation using a *larger* number of bits is found by duplicating the sign bit toward the left as many places as desired. This is called *sign extension*, and is illustrated in the following:

Length	*Representation of +1*	*Representation of −1*
8 bits	X'01'	X'FF'
16 bits	X'0001'	X'FFFF'
32 bits	X'00000001'	X'FFFFFFFF'
64 bits	X'0000000000000001'	X'FFFFFFFFFFFFFFFF'

Table 5. Examples of sign extension

We will discuss sign extension again when we examine instructions that perform shifting, and instructions that perform arithmetic on operands of different lengths.

Exercises

2.9.1.(2) Provide the 32-bit sign extensions in binary and hexadecimal notation of the five items in Exercise 2.8.2.

2.10. Binary Addition

Though number-representation details may vary slightly from one processor to another, the methods for performing binary arithmetic remain nearly the same for all processors. Thus the following is slightly more general than if only System z is discussed. The rules for adding binary digits are:

```
 0         0         1         1
+0        +1        +0        +1
 0         1         1        10 (carry)
```

Adding numbers in the *logical* representation is simplest, because all the bits are numeric digits and do not represent signs. The only unusual condition is whether or not a carry occurs out of the leftmost digit position, which would indicate whether the resulting sum is or is not correctly representable by the number of bits available.

In the two's complement representation, addition is performed in the same way, but the result is interpreted somewhat differently.

1. *All* bits of each operand are added, including sign bits, and carries out the left end of the sum are lost. (This is the same as for adding numbers in the logical representation.)
2. If the result cannot be correctly represented using the number of digits available, a *fixed-point overflow* condition occurs. The actions taken when an overflow condition occurs will vary; sometimes it can be ignored.

Using signed 4-bit binary values, we know that valid values must lie in the range $-8 \leq$ value $\leq +7$. we first add B'0010' (+2) to itself, and then we add B'0100' (+4) to itself.

```
 0010                    0100
+0010                   +0100
 0100 (no overflow)      1000 (overflow)
```

In the first case, 2+2=4, which lies in the representable range for our 4-bit numbers. But in the second case, $4+4 = -8$, because +8 is not representable. That is, the sum has *overflowed*.

A fixed-point overflow condition is possible only when adding operands of like sign: adding numbers with opposite signs always produces a representable result (or, as is often said, the result is *in range*). When an overflow occurs, the sign of the result is always the *opposite* of the sign of the two operands. The actual method used to detect overflow is simpler, since sign-change detection would require remembering the signs of both operands for comparison against the sign of the sum. Here is how it's done:

Overflow Detection Recipe

If the carries *into* and *out of* the sign bit position disagree, arithmetic overflow has occurred.

There are two kinds of binary addition: arithmetic and logical. They produce identical bit patterns, as we will see in Section 2.14. Overflow is detected *only* for arithmetic addition, while logical addition is concerned only with a possible carry out of the high-order bit position.

Exercises

2.10.1.(2)+ Consider adding the 8-bit binary number `X'F5'` to itself. There is no carry from `X'5'+X'5'=X'A'`, but there is a carry from `X'F'+X'F'=X'1E'`. Since the carry out of the low-order digit position is different from the carry out of the high-order digit position, has overflow occurred?

2.11. Binary Subtraction

Subtraction is performed by adding the two's complement of the number to be subtracted, the second operand. That is, A−B is calculated as A+(−B), where (−B) is the two's complement of B. A few examples using 8-bit binary two's complement arithmetic will help illustrate addition and subtraction.

While this prescription is essentially correct, there is a minor but important complication we'll examine after illustrating the basic scheme. (In Examples 6 and 7, note that the carries into and out of the high-order bit are different.)

- Example 1.

```
                  5-3:       0000 0101
                            -0000 0011
      becomes
                             0000 0101
                            +1111 1101
             (carry lost)    0000 0010 = 2
```

- Example 2.

```
                  3-5:       0000 0011
                            -0000 0101
      becomes
                             0000 0011
                            +1111 1011
             (no carry)      1111 1110 = -2
```

- Example 3.

```
             25-(-17):       0001 1001
                            -1110 1111
      becomes
                             0001 1001
                            +0001 0001
             (no carry)      0010 1010 = 42
```

- Example 4.

```
             (-17)-25:       1110 1111
                            -0001 1001
      becomes
                             1110 1111
                            +1110 0111
            (carry lost)     1101 0110 = -42
```

- Example 5.

```
            -17-(-25):      1110 1111
                           -1110 0111
    becomes
                            1110 1111
                           +0001 1001
            (carry lost)    0000 1000 = 8
```

- Example 6.

```
             67-(-93):      0100 0011
                           -1010 0011
    becomes
                            0100 0011
                           +0101 1101
            (no carry)      1010 0000 = -96 (overflow)
```

- Example 7.

```
             (-93)-67:      1010 0011
                           -0100 0011
    becomes
                            1010 0011
                           +1011 1101
            (carry lost)    0110 0000 = 96 (overflow)
```

- Example 8.

```
           -128-(-93):      1000 0000
                           -1010 0011
    becomes
                            1000 0000
                           +0101 1101
            (no carry)      1101 1101 = -35
```

- Example 9.

```
                  3-3:      0000 0011
                           -0000 0011
    becomes
                            0000 0011
                           +1111 1101
            (carry lost)    0000 0000 = 0
```

The above examples illustrate addition and subtraction and give the expected results. However, there is one case where the method as given above fails to detect correctly the presence or absence of overflow, and this occurs when the maximum negative number is being subtracted from something. (This is the minor complication mentioned previously.)

- Example 10.

```
             1-(-128):      0000 0001
                           -1000 0000
    becomes
                            0000 0001
                           +1000 0000
            (no carry)      1000 0001 = -127 (no overflow found?)
```

- Example 11.

```
            -1-(-128):      1111 1111
                           -1000 0000
    becomes
                            1111 1111
                           +1000 0000
            (carry lost)    0111 1111 = +127 (overflow indicated?)
```

In each of these two last cases, the result seems to be arithmetically correct, but our original overflow indication is incorrect. This is because taking the two's complement of the maximum negative number before adding it has *already* generated an overflow condition. To see how the processor can still use our overflow detection scheme as originally described (the carries into and out of the leftmost bit differ), it is worth examining the actual addition process in slightly more detail. The next section may be omitted if you are uninterested in such details, but be sure to learn the “Binary Subtraction Recipe” on page 35.

Exercises

2.11.1.(2) Give the 32-bit integer representation in hexadecimal or binary of the result of the following operations, where the operands are given as decimal numbers.

1. 10−(−10)
2. 729−65535
3. 2147483647+2
4. 1000000000+(−2147483647)
5. 0−(+0)
6. (−10)+10

Do the arithmetic in the two's complement representation, indicating for each case (1) the presence or absence of overflow, and (2) the presence or absence of a carry out of the leftmost digit position.

2.11.2.(2) Assume that the values defined in Exercise 2.8.4 are used to compute three 16-bit numbers X, Y, and Z. Using *16-bit* binary arithmetic, determine the final (hex) contents of the 16-bit fields named X, Y, and Z, and whether or not an overflow condition has occurred.

```
c(X) = c(A) - c(C)
c(Y) = c(B) + c(D)
c(Z) = c(A) + c(D)
```

2.11.3.(3) Suppose you want to subtract 1 from a binary number. A suggested technique uses these two steps: (1) change all the rightmost zeros to ones, and (2) change the previous rightmost one to zero. Create examples to show that this technique is or is not correct.

2.11.4.(4) Assume that the method in Exercise 2.11.3 is correct. How can you detect overflow conditions?

2.11.5.(2) Evaluate each of the following 32-bit sums and differences, and in each case determine (a) whether an arithmetic overflow occurs, and (b) whether there is a carry out of the leftmost bit.

1. X'7D26F071'+X'B40E99A4'
2. X'7D26F071'-X'B40E99A4'
3. X'FFFFF39A'+X'FFFE4B06'
4. X'FFFFF39A'-X'FFFE4B06'
5. X'80000003'+X'0000007C'
6. X'80000003'+X'8000007C'

2.12. How Additions and Subtractions Are Actually Performed (*)

Remember that the two's complement of a number (the two's complement representation of the negation of a number) is found by inverting each bit of the number and then adding a one in the low-order position. Digital circuits that invert bits are called *NOT* circuits. Similarly, adding a 1 bit to the low-order digit position is also easy, because each digit position of an adder circuit must add the corresponding bits of the two input operands A and B, *and* the carry bit from the next lower-order bit position, as illustrated in figure 2. as illustrated in Figure 2 on page 35.

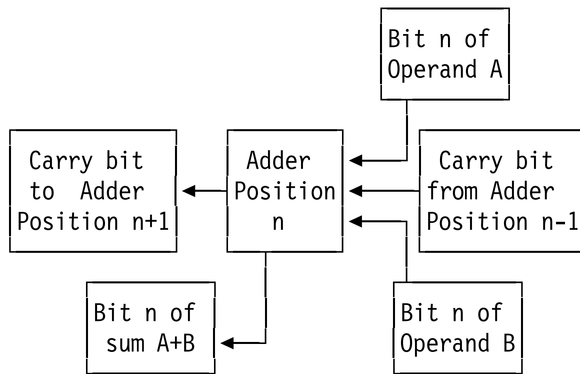


Figure 2. One stage of a binary adder

In the lowest-order position of the adder there will be no carry from a lower-order bit position. However, if an *identical* adder circuit is used, it still has a carry input that can be used to insert the 1 bit to be added to the low-order position during a complementation or subtraction operation! Thus subtraction is simply a matter of passing the second operand through a bit inverter (forming the ones' complement), and then activating the low-order carry input to the adder to add the required one-bit.

Binary Subtraction Recipe

Subtraction is performed by adding the ones' complement of the second operand and a low-order one-bit to the first operand, in a *single* operation. The subtraction in Example 10 is evaluated this way:

```
 1-(-128):         0000 0001   first operand
                  -1000 0000   second operand
    becomes
                   0000 0001   first operand
                   0111 1111   ones' complement of second operand
                  +        1   complementation bit
                (0)1000 0001   (overflow!)
```

An overflow is indicated because carries into (1) and out of the high-order bit (0) are different.

Exercises

2.12.1.(2) For each of the quantities defined in Exercise 2.8.5 on page 30, compute the following nine-bit values, indicating for each case whether or not there is a carry out of the high-order digit position, and whether or not an overflow has occurred. (Some of the values may not be representable; state which.) (1) A+C; (2) D−E; (3) Z+(−F); (4) (−E)−C; (5) (−B)+A; (6) C−Z; (7) A+(−A).

2.12.2.(3) In the ones' complement representation, subtraction is sometimes described this way:

- Take the ones' complement of the subtrahend (the number to be subtracted), and add the operands. Cross off the high-order digit and add 1 to the sum.
- If the subtrahend is greater than the minuend (the number from which the subtrahend is subtracted), take the ones' complement of the subtrahend, add the operands, then complement the result and put a minus sign in the high-order position.

Construct some examples showing how this process works, for operands of both signs and of various magnitudes.

2.13. A Circular View of Binary Arithmetic (*)

We'll use a four-bit binary representation to illustrate some concepts we have been discussing. The "circular" diagram in Figure 3 contains all 16 possible four-bit numbers.

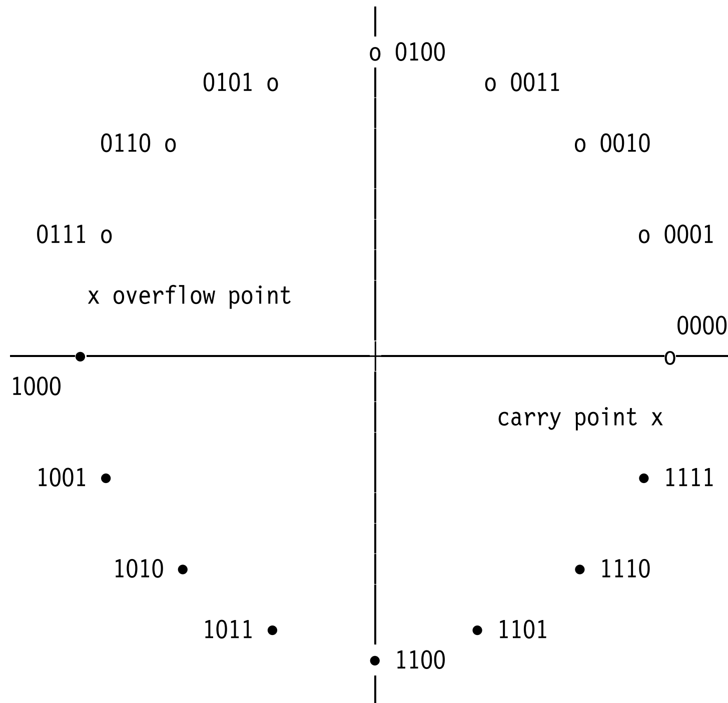


Figure 3. "Circular" representation of two's complement representation

First, suppose the numbers are considered to be the *logical* representation of the integers from 0 to 15. Counting up from 0000 by one takes us around the circle counter-clockwise from 0000 to 1111 and then back to 0000, as we would expect for numbers modulo 2^4. Adding and subtracting two numbers can be thought of as adding and subtracting the *angles* (measured counter-clockwise from 0000) represented by the numbers. Thus,

```
0100 + 0110 = 1010,    and    1100 + 0111 = 0011.
```

A carry condition occurs in addition if we go past the "carry point" in the counter-clockwise direction; similarly, a "borrow" condition occurs in subtraction if we go past the "carry point" in the clockwise direction.

For the two's complement representation, the negative of a number is the one vertically opposite it across the horizontal axis. Thus, the negative of 0011 is 1101, and the negative of 0001 is 1111. We also see that the numbers 0000 and 1000 are their own negatives, just as we found in examples 4 and 5 of Section 2.8 above.

Now, consider the numbers to be the signed 4-bit two's complement representation of the integers from −8 to +7. In the figure, the numbers with a zero sign bit are represented by open circles (o), and the numbers with a sign bit = 1 are represented by the solid black dots (•). As before, we can visualize adding and subtracting numbers by adding or subtracting the corresponding angles represented by the numbers. Now, however, we can detect overflow conditions as well: if in adding or subtracting we move in either direction past the "overflow point" between 1000 and 0111, an overflow condition occurs. Thus if we add

```
0110 + 0011 = 1001
```

we generate an overflow by passing the overflow point in a counter-clockwise direction. Similarly, in the subtraction

```
1010 - 0110 = 0100
```

we generate an overflow by passing the overflow point in a clockwise direction.

Experiment with this diagram; it reveals many properties of two's complement arithmetic.

Exercises

2.13.1.(3)+ In many early editions of the *System/360 Principles of Operation*, the Subtract operation was described as follows: “Subtraction is performed by adding the two's complement of the second operand to the first operand. All 32 bits of both operands participate. If the carries out of the sign-bit position and the high-order numeric bit position agree, the difference is satisfactory; if they disagree, an overflow occurs.”

This differs from the subtraction rule given in Section 2.13. Construct one or more examples that will show that these two descriptions are *not* precisely equivalent.

2.14. Logical (Unsigned) and Arithmetic (Signed) Results (*)

We can show that the correct algebraic result is obtained by simply adding all the bits of the operands in the two's complement representation as though they were logical operands. For 32-bit operands, the logical representation X corresponding to an arithmetic signed integer $\underline{x}$ satisfies the relation

$$X = 2^{32} + \underline{x} \pmod{2^{32}},$$

then the sum of two logical operands X and Y is

$$\begin{aligned}(X + Y) &= 2^{32} + 2^{32} + (\underline{x} + \underline{y}) \pmod{2^{32}} \\ &= 2^{32} + (\underline{x} + \underline{y}) \pmod{2^{32}} \\ &= \underline{x} + \underline{y}\end{aligned}$$

Thus the arithmetic and logical sums give the same binary result; the leftmost bits and the high-order two carry bits are just interpreted differently in the two representations.

Logical vs. arithmetic

Logical and arithmetic sums and differences of binary integers produce identical bit patterns.

We can make a further observation about adding and subtracting numbers in the logical representation. From the examples in Section 2.11, we see that in subtraction, if the second operand is *logically* smaller than or equal to the first (see examples 1, 4, 5, 7, 9, and 11) then there will be a carry out of the leftmost bit position. Conversely, we see (in examples 2, 3, 6, 8, and 10) that if the first operand is logically smaller than the second operand subtracted from it, there is *no* carry out of the left end. In these latter cases we have in some sense generated a “negative” logical answer, since the result is not correctly represented to the given number of bits. We'll see examples of these cases when we examine instructions that perform logical arithmetic.

Exercises

2.14.1.(2) Assuming an eleven-bit word, give the logical and two's complement representations of the following quantities: (1) 200, (2) 1023, (3) −1000, (4) 2047, (5) −1, (6) −1024, (7) −1023, (8) 1024, (9) −0. If a quantity is not representable, indicate that it is not.

2.14.2.(2)+ Consider the four five-bit binary numbers

```
A=11111,  B=00010,  C=10000,  D=01111.
```

For each pair of values (like A+A, A+B, etc.) determine (a) their sum, (b) whether or not a carry occurs, and (c) for arithmetic addition, whether or not an overflow occurs. Display the results in a short table. (Because addition is commutative: — X+Y = Y+X — you will need to evaluate only ten sums.)

2.14.3.(3)+ Using the same values for A,B,C,D in Exercise 2.14.2, determine the result, the carry condition, and the arithmetic overflow condition for pair-wise *subtraction* (like A-B, B-A, etc.)

of these values. Display your results in a short table; this time your table will need all 16 entries, because subtraction is non-commutative: X−Y ≠ Y−X.

2.14.4.(2)+ Can an overflow be caused by *subtracting* two numbers of opposite signs?

2.15. Examples of Representations (*)

It may help to see the differences among the sign-magnitude, radix complement (two's complement), and diminished radix-complement (ones' complement) representations.[17] All 5-bit binary numbers with positive and negative values would be represented as shown in the following table.

Binary Digits	*Logical Representation*	*Sign-Magnitude*	*Ones' Complement*	*Two's Complement*
00000	0	+0	+0	0
00001	1	+1	+1	+1
00010	2	+2	+2	+2
00011	3	+3	+3	+3
00100	4	+4	+4	+4
00101	5	+5	+5	+5
00110	6	+6	+6	+6
00111	7	+7	+7	+7
01000	8	+8	+8	+8
01001	9	+9	+9	+9
01010	10	+10	+10	+10
01011	11	+11	+11	+11
01100	12	+12	+12	+12
01101	13	+13	+13	+13
01110	14	+14	+14	+14
01111	15	+15	+15	+15
10000	16	−0	−15	−16
10001	17	−1	−14	−15
10010	18	−2	−13	−14
10011	19	−3	−12	−13
10100	20	−4	−11	−12
10101	21	−5	−10	−11
10110	22	−6	−9	−10
10111	23	−7	−8	−9
11000	24	−8	−7	−8
11001	25	−9	−6	−7
11010	26	−10	−5	−6
11011	27	−11	−4	−5
11100	28	−12	−3	−4
11101	29	−13	−2	−3
11110	30	−14	−1	−2
11111	31	−15	−0	−1

In the sign-magnitude and ones' (diminished radix) complement representations, there are two distinct representations for zero. In the two's (radix complement) representation, there is no representation for +16 corresponding to the valid representation for −16.

The sign bit in the sign-magnitude representation is attached to the (unsigned) magnitude of the value. However, the "sign bit" in the two's complement representation is not just an indicator: it is numerically significant.

Representing signed numbers in a computer always involves tradeoffs: how should "peculiar" cases like these be handled?

[17] More formally, the representation in radix r of an n-bit negative number X is $r^n - X$ in the two's complement representation, and $(r^n - 1) - X$ in the ones' complement representation.

Exercises

2.15.1.(2) Suppose your computer uses the ten's complement representation for integers. (This representation was very widely used in mechanical desk calculators, and in many early computers.) Write the following values in ten's-complement notation: (1) +28, (2) −49, (3) +527, (4) −333, (5) −1234, (6) +2469.

2.15.2.(3) Using the representations you calculated in Exercise 2.15.1, evaluate the following using ten's complement arithmetic: (a) +28+(−49), (b) +527+(−333), (c) −1234+2469.

2.15.3.(3) Write the values in Exercise 2.15.1 in the diminished radix-complement (nines' complement) representation.

Terms and Definitions

arithmetic representation
: A signed number representation.

bit
: A binary digit, taking values 0 and 1.

diminished radix-complement representation
: A signed representation where negative numbers are represented by subtracting each digit from (the radix minus 1).

hex
: See *hexadecimal*.

hexadecimal
: A base-16 representation. Its digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

logical representation
: An unsigned number representation.

ones' complement representation
: A signed binary representation where negative numbers are represented by changing each 0 bit to a 1 bit and vice versa.

overflow
: The sum, difference, product, or quotient of two numbers is too large to be correctly represented in the number of digits provided.

radix-complement representation
: A signed representation where the numerically significant high-order digit contains sign information.

sign-magnitude representation
: The familiar signed representation of numbers with prefixed + or − signs.

two's complement representation
: A signed binary representation where the high-order bit contains sign information, and has weight -2^{n-1}.

Chapter II: System z

```
IIIIIIIIII   IIIIIIIIII
IIIIIIIIII   IIIIIIIIII
    II           II
    II           II
    II           II
    II           II
    II           II
    II           II
    II           II
    II           II
IIIIIIIIII   IIIIIIIIII
IIIIIIIIII   IIIIIIIIII
```

This chapter's three sections introduce the main features of System z processors:

- Section 3 describes basic structures: memory organization and addressing, general purpose registers, the Program Status Word (PSW), and other topics.
- Section 4 discusses the instruction cycle, basic machine instruction types and lengths, exceptions and interruptions and their effects on the instruction cycle.
- Section 5 covers address calculation, the "addressing halfword", Effective Addresses, indexing, addressing problems, and virtual memory.

3. Conceptual Structure of System z

```
 3333333333
333333333333
33        33
          33
          33
       3333
       3333
          33
          33
33        33
333333333333
 3333333333
```

We can describe the structure of most computers in terms of four functional units: memory, arithmetic, control, and input-output. A real computer may not identify components this way, but it helps us to think of them as distinct units.

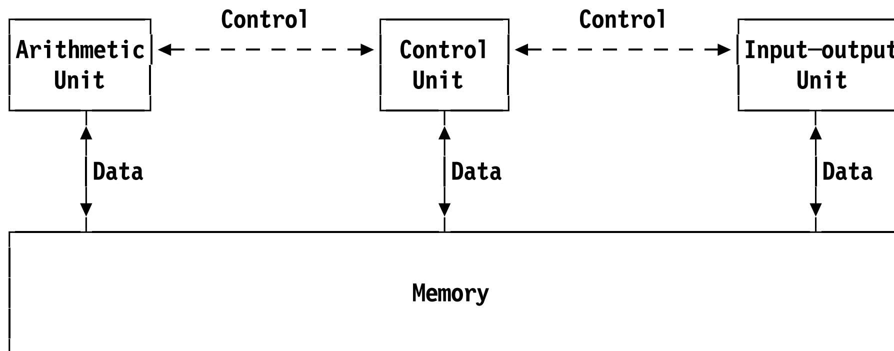


Figure 4. Conceptual structure of a typical computer

The solid lines in Figure 4 represent data paths among the various units, and the dashed lines indicate the flow of control signals. As indicated, the *same* memory holds instructions for the control unit and the data used by the arithmetic and input-output units. This gives modern digital processors their flexibility and power: they can treat instructions as data or data as instructions.

System z makes no special distinction between the arithmetic and control units, and the combination is often called the “Central Processing Unit”, or “CPU”.

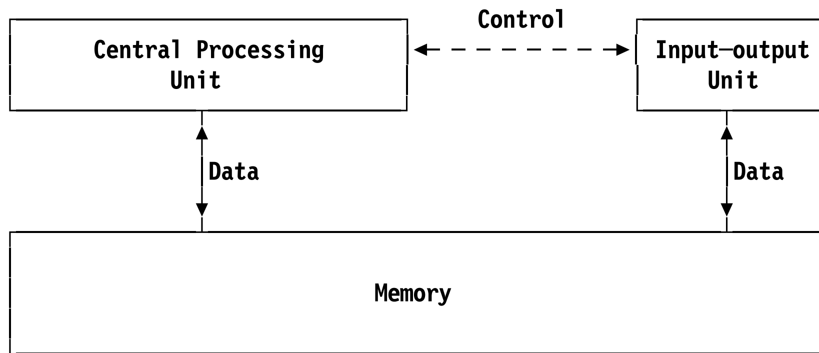


Figure 5. Conceptual structure of System z

"Memory" is sometimes called "central storage" or similar terms. It refers to that part of the processor holding the directly accessible instructions and data to be manipulated by those instructions.

As Figure 5 indicates, input and output — once initiated by the CPU — is performed between external devices and memory, and does not pass through the CPU. The Input-output Unit communicates the status of its operations to the CPU, indicating error conditions or completion of the operation.

3.1. Memory Organization

Digital computers deal with data consisting of binary digits, easily and rapidly accessed from "central memory". The basic data item is an eight-bit group called a *byte.*[18] The bits in a byte are numbered from 0 to 7, beginning on the left with the numerically most significant digit. (The importance of designating the "left" side of a byte will be clearer when we consider *groups* of bytes.) In Figure 6, the leftmost bit is a 1-bit, and the rightmost bit is a 0-bit.

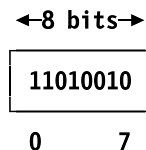


Figure 6. A byte containing 8 binary digits

Bytes in memory are arranged so that each byte may be referenced as easily as any other. The bytes are individually numbered beginning at zero; the number associated with each byte is called its *memory address*. Memory may be thought of as a linear string of bits; the bits are grouped into bytes arranged in order of increasing addresses. Only bytes have addresses; bits within a byte don't have their own addresses.

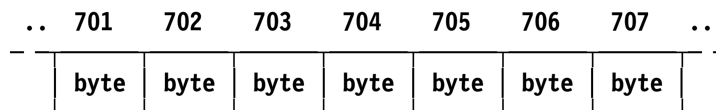


Figure 7. A portion of memory, with addresses shown above each byte

The bits in a byte are accessed (or "read") by the CPU without being changed. Reading the contents of a byte does not affect the contents; the memory provides the CPU with a *copy* of the

[18] Because the eight bits in a byte are often described using two hex digits, some people like to call a "half byte" hex digit a cute name like "nibble" or even "nybble". We won't.

contents of a byte. Storing (or "writing") a new bit pattern into a byte replaces the previous contents.

Many machine instructions referring to memory actually refer to a group of consecutive bytes. In such situations the group is normally addressed by referring to its *leftmost* member, the byte with the lowest address in the group.[19] Also, some instructions require the address of a group of bytes (the address of the leftmost byte) to *also* be a multiple of the length of the group, in which case we say that the group is *aligned*.[20] The possible lengths for such groups of bytes are 2, 4, 8, or 16; we sometimes refer to them as halfwords, words (or fullwords), doublewords, and quadwords respectively.

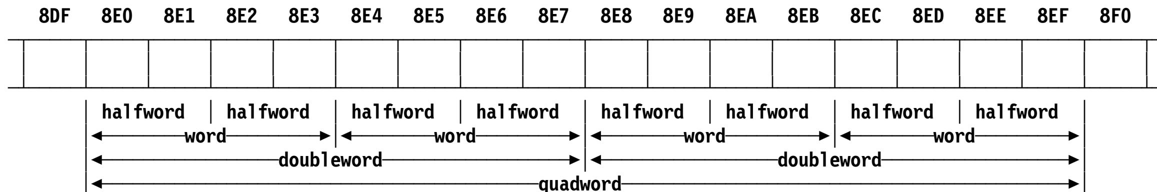


Figure 8. A portion of memory

When some operation manipulates a group of bytes, we call that group an "operand": something that is "operated on". The group always consists of data from consecutively-addressed bytes in memory.

Some operations treat the operand as a string of bits whose meaning for that operation is independent of the fact that they are arranged into 8-bit bytes in memory. For example, suppose a halfword operand (a group of two bytes whose address is divisible by 2) is specified for an operation, and the address of the 16-bit operand is `X'8EA'`. Then the 16 bits in the bytes at `X'8EA'` and `X'8EB'` will be treated as a *single* 16-bit halfword, and we ignore the fact that they are stored in memory as two distinct eight-bit bytes. Thus, bit 0 of the halfword operand — its leftmost bit — corresponds to bit 0 of the byte at `X'8EA'`, and bit 15 of the halfword operand — its rightmost bit — corresponds to bit 7 of the byte at `X'8EB'`.[21]

Bytes in memory contain *only* bit patterns. Whether the bit pattern is interpreted as an instruction, or as one of many types of data, depends only on the context of its use; at one time it may be data, and at another, an instruction. Whatever the interpretation, however, a byte is simply a group of eight bits.

We now see why we use hexadecimal (base 16) notation for expressing binary numbers instead of octal (base 8) notation. It is simplest to arrange bits in groups of the same size, and the presence of eight bits in a byte makes four-bit groups natural. A half-byte contains 4 bits, exactly the number of bits needed to represent one hex digit. If octal notation is used, a byte would contain two three-bit octal digits and two extra bits.

Exercises

3.1.1.(2)+ An area of memory reserved for data begins at address `X'2EC9'` and ends with address `X'30A6'` (including the start and end bytes!). How many bytes are there in the area, and how many halfwords, words, and doublewords can be stored in the area?

3.1.2.(1) The memory of System z can be thought of as a continuous string of bits. Does each individual bit in memory have an address? Explain.

---

19 This is true with few exceptions, which we will note as they appear. For now, remember "leftmost" as the rule.

20 In early System/360 processors, many memory operands had to be aligned on byte boundaries whose addresses were a multiple of the operand's length. While this is no longer required for most (but *not* all) instructions, proper alignment is always a good programming practice.

21 z/Architecture processors use what is called "big-endian" addressing; we'll examine "endianness" in detail in Chapter VII, Section 26.7.

3.1.3.(2) Suppose we are interested in the string of contiguous bits starting with bit 5 of memory address X'1A023' and ending with bit 1 of the byte at memory address X'1A03B' (including the start and end bits). Determine the number of bits in the string.

3.1.4.(1) State which of the following addresses refer to halfwords, words, and doublewords: (1) X'123456'; (2) X'234567'; (3) X'345678'; (4) X'000BBC'.

3.1.5.(1) Determine the number of bits that can be stored in a memory area of the following sizes: (1) X'20000' bytes, (2) X'8000' bytes, (3) X'200000' bytes.

3.1.6.(1) Express the contents of the byte in Figure 6 on page 43 in octal notation and in hexadecimal notation.

3.1.7.(1)+ If you examine the rightmost hex digit of a memory address, what can you tell about the alignment of the address?

3.2. Central Processing Unit

The CPU performs the operations specified by your program. An important element of the CPU is a set of *registers*, a special and very fast form of memory kept very close to the instruction and data processing functions of the CPU.

- The general registers are used for arithmetic and logical operations, and to hold addresses of data and instructions;[22]
- the Floating-Point Registers are used for floating-point arithmetic and data;
- the Program Status Word is used by the CPU to control the progress of your program as it is executed.

3.3. General Registers

There are sixteen general registers, numbered from zero to fifteen. Each is 64 bits (or 16 hex digits or 8 bytes) long. They are represented schematically in Figures 9 and 10.

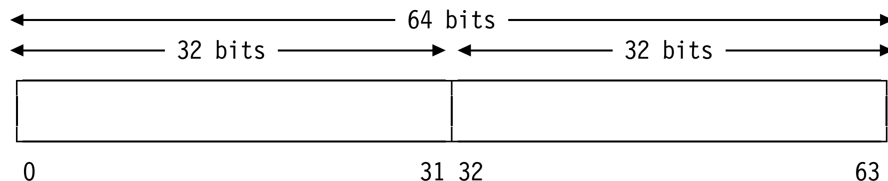


Figure 9. A single 64-bit general register

When we discuss instructions that do 32- and 64-bit arithmetic, we'll understand why this picture shows two 32-bit parts of a 64-bit general register.

[22] Because the general registers are used for so many activities, they are sometimes called "General Purpose Registers".

General Register 0	General Register 1
General Register 2	General Register 3
General Register 4	General Register 5
General Register 6	General Register 7
General Register 8	General Register 9
General Register 10	General Register 11
General Register 12	General Register 13
General Register 14	General Register 15

Figure 10. All sixteen general registers

This figure arranges the registers in pairs, the left register being even-numbered and the right being the next higher odd-numbered register. Some operations require using a *pair* of registers, and in such cases it is *always* an even-odd-numbered pair.

We will often refer to the general registers using a short notation: we sometimes write "GRn" (meaning the rightmost 32 bits of a 64-bit register) or "GGn" (meaning all 64 bits) or simply "Rn" to refer to general register n when the register length is clear from context. Thus, in Figure 10, we might use R1 to mean register 1, R14 to mean register 14, and so on.

Be Careful!

"R1" (without a subscript) is not the same as the notation "R_1" (with a subscript). This difference will be important when we discuss machine instructions.

Exercises

3.3.1.(1) Suppose a shifting operation requires the use of a pair of general registers. Is it possible to perform the shifting operation using both GR7 and GR8? Using GR15 and GR0? Using GR6 and GR7?

3.3.2.(1) How many bytes can be placed in a pair of general registers?

3.4. Floating-Point Registers

On the earliest System/360 models only four floating-point registers were available, and then only as an option. Sixteen are always present in System z processors, as we will see in Section 32.7. Each is 64 bits (16 hex digits, 8 bytes, or 1 doubleword) long. We will look into this more deeply when we discuss floating-point instructions and data in Chapter IX.

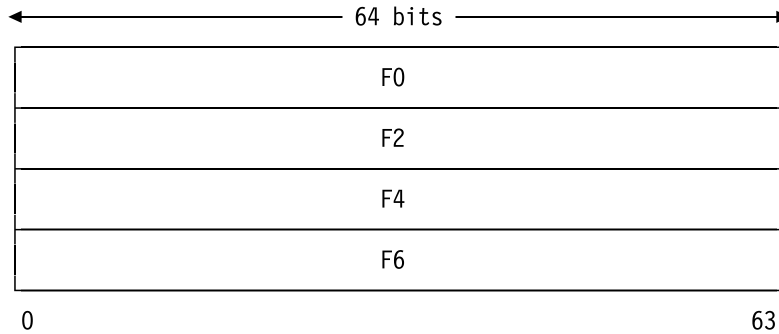


Figure 11. Four Floating-Point Registers

Sometimes the floating-point registers contain operands 32 bits long. In this case they use only the *left* half of the register, and the rightmost 32 bits are ignored. In other situations, a floating-point instruction using 128-bit operands will use a *pair* of floating-point registers.

We won't mention the floating-point registers until we discuss instructions for floating-point arithmetic. We sometimes use the abbreviations “FPRn” or “FRn” or “Fn” to refer to floating-point registers.

In some cases we use “register” to describe a general register or a floating-point register (or some other type of register); which is meant will be clear from context.

Exercises

3.4.1.(1) How many *short* (32-bit) floating-point numbers can be held in a floating-point register?

3.4.2.(3) Can you think of any reasons why the designers of System/360 and System z included a *separate* set of registers for floating-point arithmetic? That is, why should it not be possible to use the general registers for binary integer arithmetic, addresses, *and* floating-point arithmetic?

3.5. Program Status Word (PSW)

Usually, the Program Status Word (PSW) is of no immediate concern, and you need not worry about its contents. It is another internal register that contains various fields indicating the status of the program being executed. As the System/360 and System z processors have evolved, the PSW has taken several forms.

For our purposes, we need to know about only a few parts of a PSW: the Instruction Address (IA), the Condition Code (CC), the Instruction Length Code (ILC), and the Program Mask (PM). Of these, the IA is most important now; we'll see more about the others later.

Figure 12 illustrates these four parts of a PSW (and the “System Flags”). The IA is always in the rightmost position; the positions of the other three aren't significant. (In fact, PSWs since about 1975 no longer have a field for the ILC.)

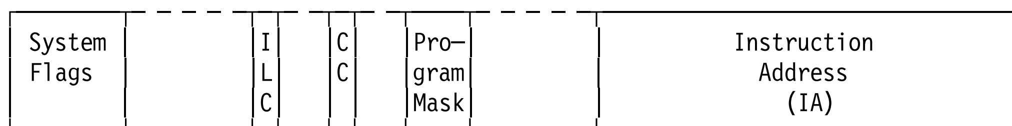


Figure 12. Sketch of a Program Status Word

The PSW for the currently executing program resides in the CPU, *not* in memory.

The CC is *set* (given a value) by some instructions — for example, to indicate that the result of an addition operation is a negative number. Other instructions may have no effect on the CC; in

such cases we say that it is *not set,* or that its value is *unchanged.* Still other instructions can test the value of the CC and make decisions based on the result.

Among the system flag bits in the PSW is the “P” bit, which determines whether or not the CPU will allow certain instructions to be executed. If the “P” bit is 1, the CPU is in *Problem State* and will not execute *privileged* instructions, such as those specifying Input-Output operations. If you try to execute a privileged instruction while the CPU is in Problem State, a program interruption will occur instead. When the “P” bit is 0, the CPU is in *Supervisor State*, and it allows any instruction to be executed. This is how supervisory programs retain control over activities critical to the smooth operation of a complex programming system.

3.6. Other Registers

In all System z processors, the CPU contains many additional registers including *Access Registers* and *Control Registers*. The Access Registers are used for special types of addressing. The Control Registers are not normally available to application programs: they are not used for arithmetic or for addressing by a program because they control various execution functions.

We'll say more about these and other registers as needed.

3.7. Input-Output (I/O)

Data transmission between main memory and external devices is managed by *channels.* Channels transmit bytes of data from an external device to memory, or from memory to an external device, while allowing the CPU to continue with the execution of a processing program. We will use some simple forms of I/O later, especially for Programming Problems at the end of each chapter.

3.8. Features, Facilities, and Assists

The System z family of processors has grown from its original System/360 capabilities.[23] The added capabilities are sometimes called “features”, “facilities”, or “assists”. For example, the “long-displacement” facility is a recent addition. We assume that your CPU has all the facilities needed to execute our instructions and program examples.

3.9. Microprograms and Millicode (*)

For the earliest System/360 models, internal operations were controlled by “microprograms” that were kept in a special type of read-only memory. The internal circuits were then “programmed” by a combination of hardware and micro-instructions to act like a System/360 processor!

Modern processors, in contrast, use a combination of hardware, microcode, and “millicode” instructions to execute the instructions you write, and to perform other CPU “housekeeping” functions. Millicode instructions are kept in a reserved area of main memory. They are very similar to your instructions, but can do things that your “normal” instructions can't do.[24] The set of millicode instructions is sometimes referred to as “firmware”.

Terms and Definitions

byte
A group of 8 bits; the basic addressable unit of memory.

---

[23] This is quite an understatement.

[24] If you're interested in learning more about millicode, see the article by Lisa Heller and Mark Farrell in the *IBM Journal of Research and Development*, Vol. 48 No. 3/4, May/July 2004.

CC
: Condition Code, a 2-bit field in the PSW used to indicate the status or result of executing certain instructions.

CPU
: Central Processing Unit

FPR
: Floating-Point Register

GR
: General Register

ILC
: See *Instruction Length Code*.

Instruction Length Code
: A 2-bit field in low storage indicating the length in halfwords of an instruction that caused a particular type of interruption.

millicode
: Internal instructions used by the CPU to perform operations too complex to be done cost-effectively in hardware.

problem state
: A state in which the CPU disallows the execution of certain *privileged* instructions.

PSW
: Program Status Word, containing information about the current state of a program.

supervisor state
: A state in which the CPU allows the execution of all instructions.

4. Instruction Execution

```
        44
       444
      4444
     44 44
    44  44
   44   44
  44444444444
 444444444444
        44
        44
        44
        44
```

In this section we see how instructions are executed by the CPU, and then look at examples of the formats used for five representative classes of instructions.

As we saw in Figure 5 on page 43, instructions executed by the computer reside in memory along with the data to be processed. Instructions in System z can be 2, 4, or 6 bytes long. Instructions are *always* aligned so that the leftmost byte is on a halfword boundary: that is, the address of an *instruction* must always be divisible by two. This alignment does not depend on the length of the instruction; it doesn't matter, for instance, that a 4-byte instruction begins halfway between two word boundaries. It is more precise to say that instructions are 1, 2, or 3 halfwords long.

Unlike some types of data, there is no requirement that an instruction start at an address that is a multiple of its length; only that it start on a halfword boundary.

4.1. Basic Instruction Cycle

The process of executing instructions may be visualized in Figure 13.

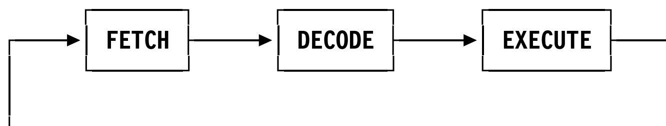


Figure 13. Basic instruction cycle

In the "fetch" portion of the cycle, the CPU locates the instruction beginning at the halfword in memory whose address is contained in the rightmost part of the PSW (the Instruction Address, or IA), and places it into an internal register where it is decoded. Though this internal register is not accessible to programs, we will refer to it as the Instruction Register, or IR. The CPU determines the length of the instruction by examining its first two bits, as we will see shortly.

To complete the fetch portion of the cycle, the CPU adds the length in bytes of the instruction now in the Instruction Register to the IA in the PSW, so that the IA will contain the address of the *next* instruction to be fetched when the current instruction completes its execution. This

means that instructions must be packed tightly in memory; there are no leftover bytes or gaps between instructions executed in sequence.

To decode the instruction, the CPU examines the bit pattern in the IR to see what action is intended. Since (1) the bytes were brought from memory and (2) the memory contains both data and instructions, the bytes brought to the IR may actually represent data and not instructions. The CPU has no way of knowing this; it simply goes to the memory address in the IA portion of the PSW and fetches those bytes into the IR to be interpreted as an instruction. If this is what you intended, good; otherwise, strange things can happen.

Not all of the possible bit patterns in the IR might represent “valid” instructions (i.e., actions the CPU can execute, or will allow to execute). The decoding mechanism can sometimes detect confused situations (such as data being interpreted as instructions) before too much damage has been done, and cause remedial actions to be initiated.

Assuming that the bytes in the IR contain a valid instruction, further actions may be necessary before the decoding is completed, such as calculating addresses of the operands to be manipulated during the execute portion of the cycle.

During the execution phase, the actual operation is performed. It could cause the contents of one general register to replace the contents of another, or it may involve many intermediate steps of complicated logic or arithmetic. If no errors are detected during the execution phase (such as attempting to divide a number by zero), the CPU resumes the instruction cycle by returning to the fetch portion of the cycle.

We sometimes refer to the entire cycle of fetching, decoding, and executing an instruction simply as “executing” that instruction.

Instructions

The IA portion of the PSW addresses the next instruction to be fetched. If you didn't intend the fetched bytes to be an instruction, it's a mistake you must correct.

Exercises

4.1.1.(2) How could you build a CPU without a separate Instruction Address (such as in the z/Architecture PSW)?

4.2. Basic Instruction Types

The instructions provided by the original System/360 processors had five formats:

1. register-and-register (RR)
2. register-and-indexed-storage (RX)
3. register-and-storage (RS)
4. storage-and-immediate (SI)
5. storage-and-storage (SS)

Modern System z processors support over 30 instruction formats that we'll introduce as needed. These five formats are enough for now, because newer instruction formats are variations on these basic forms.

The letters RR, RX, RS, SI, and SS are abbreviations that indicate the type, or class, of an instruction. Individual instructions belonging to each class will be treated in later chapters.

Figure 14 on page 52 gives a useful way to visualize the behavior of these classes:

- RR-type instructions operate on data within registers;
- RX- and RS-type instructions operate on data between registers and memory;

- SS-type instructions operate on data in two memory locations; and
- SI-type instructions operate on data in memory using an operand in an instruction.

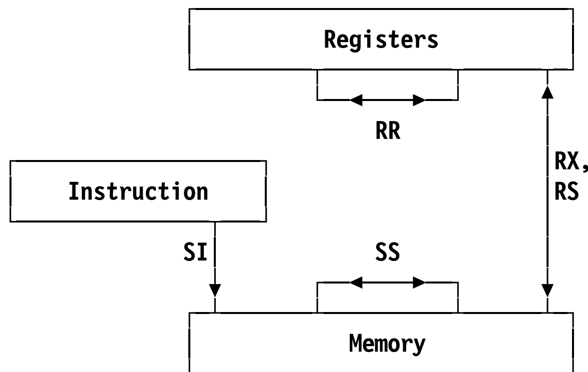


Figure 14. Instruction formats and data interactions

The first byte of an instruction always contains an *operation code* (often abbreviated "opcode"), specifying the operation to be performed. The second byte usually contains data about the location, type, or length of the data to be operated on. This second byte has several forms: it is called the "register specification" byte (for RR, RX, and RS instructions), the "immediate data" byte (for SI instructions), or the "length specification" byte (for SS instructions).[25] The interpretation of this second byte therefore depends on the class to which the instruction belongs.

- RR-type instructions are always one halfword long.

operation code	register specification

Table 6. RR-type instruction format

- RX- and RS-type instructions are always two halfwords long.

operation code	register specification	addressing halfword

Table 7. RX-type and RS-type instruction format

The RX- and RS-type instruction formats differ only in the interpretation of the bits in the "Register Specification" byte.

- SI-type instructions are always two halfwords long.

operation code	immediate data	addressing halfword

Table 8. SI-type instruction format

Instead of a register specification, the second byte of an SI-type instruction contains an 8-bit data item used in executing the instruction.

[25] In some newer instructions, the second byte may contain another part of the opcode; and in some instructions, part of the opcode may be in the *sixth* byte! The CPU knows, so you needn't worry.

- SS-type instructions are always three halfwords long.

operation code	length specification	addressing halfword	addressing halfword

Table 9. SS-type instruction format

For most instructions except RR-type instructions, an *addressing halfword* is used by the CPU to compute the address of an operand; this important process is described in "5.1. The Addressing Halfword", on page 62, and again in Section 20. These classifications are *not* exhaustive; many newer instructions are variations on these basic forms.

Exercises

4.2.1.(1) Must a 4-byte RX-type instruction begin on a word boundary?

4.2.2.(1) What is the length of the shortest instruction in System z?

4.2.3.(2) How is it possible for instructions of different lengths to be packed tightly into memory with no wasted bytes?

4.2.4.(1)+ May an instruction begin on a word boundary? On a doubleword boundary?

4.2.5.(2)+ Figure 14 on page 52 implies that both instructions and data reside in the same memory. How can you tell if a given string of bytes represents instructions or data?

4.3. Instruction Lengths

The first two bits of the operation code tell the CPU how many bytes to fetch from memory. Since at least two bytes per instruction must always be fetched, the CPU can check the two leading bits to tell how many more bytes (if any) are required. The bit patterns are shown in Figure 15; "xxxxxx" represents the remaining six bits of the eight-bit operation code.

```
00xxxxxx   2-byte instructions such as RR-type
01xxxxxx   4-byte instructions such as RX-type
10xxxxxx   4-byte instructions such as RS- and SI-type
11xxxxxx   6-byte instructions such as SS-type
```

Figure 15. Opcode bit patterns for typical instruction types

If the first two bits of the opcode are 00 the instruction is one halfword long; if the bits are 01 or 10 it is two halfwords long; and if the bits are 11 it is three halfwords long.

Before decoding the instruction, the CPU places the number of *pairs* of bytes in the instruction (the number of halfwords: 1, 2, or 3) into an internal two-bit PSW field called the Instruction Length Code (ILC). It is important to remember that the two bits of the ILC are *not* the same as the first two bits of the opcode. Table 10 on page 54* shows the relationship between the first 2 bits of the opcode and the ILC:

* Courtesy of Michael Stack.

ILC (decimal)	ILC (binary)	Instruction types	Opcode bits 0-1	Instruction length
0	B'00'			Not available
1	B'01'	RR	B'00'	One halfword
2	B'10'	RX	B'01'	Two halfwords
2	B'10'	RS, SI	B'10'	Two halfwords
3	B'11'	SS	B'11'	Three halfwords

Table 10. Instruction Length Code and instruction types

If an error is detected during decoding or executing the instruction, the PSW at the time of the error is saved, and the programmer can examine the ILC and the IA of the saved PSW to determine what instruction caused the error. If the ILC was not saved it would not be possible to determine the exact location of the offending instruction, since the location of the *next* instruction to be executed is already in the IA portion of the saved PSW, and the length of the bad instruction could have been 2, 4, or 6 bytes.

Exercises

4.3.1.(1) Is it possible for a six-byte instruction to be mistaken by the CPU for a four-byte instruction? Explain.

4.3.2.(2)+ A program segment consists of the following six operations (only the opcodes are given): `X'05'`, `X'58'`, `X'89'`, `X'5A'`, `X'D2'`, `X'50'`. Determine the length in bytes of the program segment.

4.3.3.(2) For each of the instructions in the previous exercise, determine the value of the Instruction Length Code after each has been fetched.

4.3.4.(2) By examining Figure 15 on page 53, deduce a simple formula that can be used to determine, for any System z instruction, what number should be added to the Instruction Address in the PSW to give the address of the following instruction.

4.3.5.(2)+ Make (and study) a short table of four rows, with the following column headings: (1) value of first two bits of opcode, (2) instruction type, (3) instruction length, (4) ILC after instruction fetch is complete, and (5) number of addressing halfwords.

4.3.6.(2)+ The following twelve halfwords taken from memory are known to be a sequence of instructions. (The spaces have been inserted for readability; the bytes in memory are contiguous.)

```
90EC D00C 0580 50D0 89EA D703 89EE 89EE 18CD 41D0 89E6 1B11
```

Determine (1) how many instructions there are, (2) their lengths, and (3) their types.

4.3.7.(3)+ Suppose you know the PSW and ILC after an execution error has occurred. How do you determine the address of the instruction that caused the error?

4.3.8.(2) What would happen if gaps are left between instructions?

4.4. Some Operation Codes (*)

Table 11 on page 55 summarizes the characteristics of some basic instructions, as they depend on the first *four* bits of the operation code. As described above, the first two bits determine the type and length of the instruction. The second pair of bits determines (to some degree) the operand length or the general functions performed by the instructions. (These groupings are only approximate, but they may help you to appreciate how opcodes are designed.)

A closer examination of a complete table of operation codes reveals a great deal of symmetry in the opcodes used for similar functions. For example, the four original System/360 instructions

that perform the “Logical AND” operation all have operation codes where the second hex digit is 4 and the first hex digits differ by multiples of 4 (X'14', X'54', X'94', and X'D4').

First pair of bits	**Second pair of bits**			
	00	**01**	**10**	**11**
00 (RR)	Branching, status switching	Word logical, fixed-point binary	Long hexadecimal floating-point	Short hexadecimal floating-point
01 (RX)	Branching, halfword fixed-point	Word logical, fixed-point binary	Long hexadecimal floating-point	Short hexadecimal floating-point
10 (RS, SI)	Branching, shifting, status switching	Fixed-point, logical, I/O		Logical
11 (SS)		Logical		Packed decimal

Table 11. General instruction classifications

Since we will refer to instructions almost entirely using mnemonics — short abbreviations for their full names — these details are only of minor interest.

Exercises

4.4.1.(2) Examine the operation codes given in Exercise 4.3.2, and determine their general instruction classifications from Table 11.

4.5. Interruptions (*)

The instruction cycle shown in Section 4.1 on page 50 describes the basic mechanism of instruction sequencing. However, a more workable view requires understanding *interruptions,* sometimes called *interrupts.* We'll discuss them briefly here, and in more detail when we describe possible exceptions caused by instructions.

When an interruption occurs, the CPU stores the PSW that currently controls its operation in a predefined area of memory, and immediately replaces it with a new one from a different predefined area of memory. Many things can cause this PSW switching: a program may contain an instruction that causes an interruption to occur, or some external event such as a completed I/O operation could cause an interruption. The basic mechanism used for handling interruptions is illustrated in Figure 16.

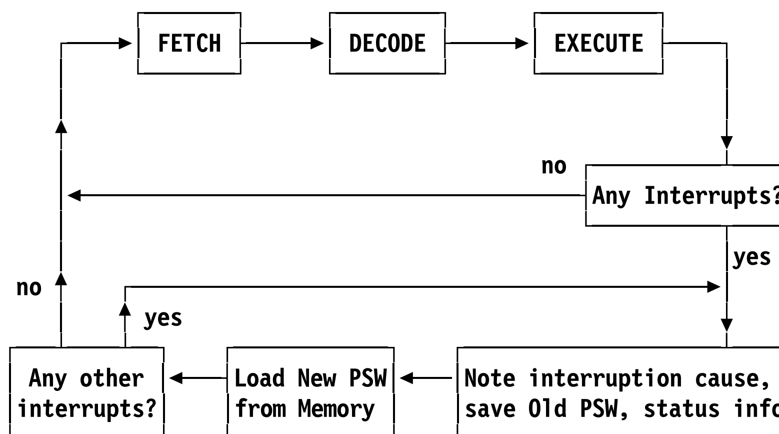


Figure 16. Instruction cycle with interruptions

The usual cycle of fetching, decoding, and executing will continue undisturbed so long as no interruption occurs.[26] When an interruption condition *is* present, the CPU first examines bits in the PSW (or in the Program Mask or in other special registers) to see whether the interruption should be accepted. If these bits are zero, the interruption condition is said to be *masked* or *disabled*, and the CPU takes a default action before proceeding to the next instruction.

If the interruption is not masked (or is *enabled*), the CPU places information about the cause of the condition into a reserved “Interruption Code” area near the low-address end of memory. The CPU then stores the current (old) PSW and loads a new PSW. Instruction fetching then resumes, with the next instruction being fetched from the memory address specified by the IA portion of the newly-loaded PSW. This will almost always be in the Supervisor.

Normally, the new PSW will disable further interruptions until the Supervisor can save information about the status of the program being interrupted. After this status information (such as register contents and the old PSW) has been saved, the CPU can be enabled for further interruptions. After the interruptions have been handled, the saved status information is restored and the interrupted program can be resumed.

These are the six *classes* of interruptions, with examples of possible causes:

1. Restart (operator action)
2. External (timer, clock comparator)
3. Machine Check[27] (equipment malfunction)
4. Input-Output (an I/O device has signaled a condition)
5. Program (exception condition during program execution)
6. Supervisor Call (program requests an Operating System service)

Corresponding to each class is an area of memory where an old PSW is stored, and an area from which a new PSW is loaded by the CPU. Thus there are six areas in memory into which old PSWs are stored, and another six areas from which new PSWs are retrieved. These areas are at fixed positions in the low-address end of memory; a programmer has no control over where they are placed.

We sometimes distinguish two different classes of interruption. The first is caused by events whose occurrence cannot be predicted, or for which a program cannot test in advance: these are sometimes called *involuntary* or *asynchronous* interrupts. The first four classes of interruption are involuntary. Except for the restart interruption, all the involuntary interruptions can be masked.

The program and supervisor call interruptions are *voluntary* or *synchronous*. They are mutually exclusive, and cannot both occur at the same time. Program interruptions are caused by many conditions, as you will discover. A supervisor call interruption occurs only as a result of executing a Supervisor Call (SVC) instruction.

The program and supervisor call interrupts are “voluntary” because the program can (if it wishes) know what instruction will be executed next, and what interruption-causing actions that instruction could take.

4.6. Exceptions and Program Interruptions (*)

Programs can create many types of exception condition. Some of them may not be serious, and your program can tell the CPU to take some default action (like setting the Condition Code, or generating a specified default result). Other exception conditions require interrupting the instruction cycle.

We will be most concerned with *program interruptions*. They may be caused by error conditions detected during any of the three portions of the instruction cycle. For example, if the IA specifies

---

[26] Figure 16 doesn't account for the possibility that an interruption can occur during the fetch or decode phases. In almost all cases, this distinction is unimportant.

[27] This interruption shouldn't be masked off because the CPU must save diagnostic information before the situation gets worse.

that an instruction should be fetched from an odd memory address, no fetch occurs and an interruption is generated instead. During the decode phase, the CPU may discover that the operation code is invalid. Similarly, an error condition such as attempting to divide a number by zero may occur during the execution phase.

Exceptions and Interruptions

Exception: An unusual condition possibly requiring attention; your program may be able to request the CPU take a default action and continue execution, or cause an interruption.

Interruption: An exception condition requiring alteration of the normal sequence of program execution by passing control to the Operating System.

For most program interruption conditions, the Operating System provides a brief indication of the cause of the interruption. Additional diagnostic information may also be given, such as the old PSW and the contents of the general and floating-point registers, and the contents of various areas of memory. You can then use this information to try to deduce the cause of the interruption.

The most common types of program interruptions are shown below with their associated Interruption Codes. This list is not complete, but may help you find the causes of typical interruptions generated by your programs.

IC=1 Invalid Operation Code. The decoding phase has found an operation code that cannot be executed. This could be due to (1) allowing data to be fetched as instructions, or (2) the program's destroying part of itself.

IC=2 Privileged Operation. The program is trying to execute an instruction not allowed in problem state.

IC=3 Execute exception. An execute instruction is attempting to execute another execute instruction.

IC=4 Access, Protection. The program has attempted to refer to some area of memory to which access is not allowed. There can be other causes, but this is the most common.

IC=5 Addressing. The program has attempted to address a nonexistent memory address.

IC=6 Specification Error. This can be caused by many conditions, but a common cause is referring to an odd-numbered register when an even-numbered register is required. An odd IA in the PSW indicates an attempt to access an instruction not starting on a halfword boundary.

IC=7 Data Exception. This is caused by invalid packed decimal data, or by binary or decimal floating-point conditions described in Chapter IX.

IC=8 Fixed-Point Overflow. This is caused when a fixed-point binary result is too large.

IC=9 Fixed-Point Divide Exception. A binary divide instruction has found that a quotient would be too big to fit in a register, or a divisor is zero.

IC=A Decimal Overflow. A packed decimal result is too large to fit in the result field.

IC=B Decimal Divide. A packed decimal quotient is too large to fit in the result field, or a divisor is zero.

IC=C Hexadecimal floating-point exponent overflow. A hexadecimal floating-point result is too large.

IC=D Hexadecimal floating-point exponent underflow. A hexadecimal floating-point result is too small.

IC=E Hexadecimal floating-point lost significance. A hexadecimal floating-point result has lost all its significant digits.

IC=F Hexadecimal floating-point divide exception. A hexadecimal floating-point operation is attempting to divide by zero.

Four of the fifteen possible program interruption conditions are often regarded as harmless: fixed-point and decimal overflow exceptions, and hexadecimal floating-point exponent underflow and

lost-significance exceptions. By setting an appropriate mask bit in the Program Mask to zero (see Figure 12 on page 47), you can use the SPM instruction (described on page 234) to request that the CPU take a predefined default action and continue execution without causing an interruption. Other default actions can be requested for many floating-point operations, by setting mask bits in the Floating-Point Control Register (more about this in Chapter IX).

Thus, exception conditions can sometimes cause an interruption, and sometimes take a default action if the interruption is masked. For example, a fixed-point overflow if enabled will cause an interruption with interruption code 8; but if masked off, the CPU will set the Condition Code to 3 before fetching the next instruction.

The CPU may seem overly cautious about detecting error conditions: the number of ways to generate interrupts sometimes seems larger than the number of ways to write a correct program! However, these error-detection mechanisms help catch program errors: an interruption condition will usually be generated before your program has gone too far, and you will have an indication that something is wrong before the cause is obscured.

Consider the problem of finding program errors on a CPU in which all bit patterns represent valid data or operation codes, and where none but the most unusual error conditions were caught. The processor could offer little help, and you would have to write programs with many internal checks and tests. In addition to the extra effort needed to write correct programs, the time used for checking would cause the program to run more slowly. Program interruptions should be seen as helpful clues from the CPU, and not as an indication that something is wrong with the processor.

Exercises

4.6.1.(2)+ Suppose the contents of the following 8-byte System/360 PSW[28] (sketched in Figure 12 on page 47) was displayed as the result of a program interruption. What error condition is immediately evident? (The “xxxxxxxx” digits are unimportant for this exercise.)

```
xxxxxxxx 4017E26F
```

4.6.2.(3) Suppose the 8-byte Program New PSW area of memory had been initialized with the following “New PSW”: (The “xxxxxxxx” digits are unimportant for this exercise.)

```
xxxxxxxx 0000A237
```

What do you suppose would happen if any program interruption occurs?

4.6.3.(1) What caused the following Interruption Codes?

1. 0001
2. 0009
3. 000C

4.7. Machine Language and Assembler Language

Sometimes people refer to Assembler Language programming as “machine language” or “processor language” programming. In the earliest days of digital computers, there were almost no programming tools like assemblers and compilers, so the instructions and data for programs had to be created in the form of binary (or decimal or hexadecimal) digits that were loaded directly into memory for execution, without any intermediate translation.

Thus, we consider “machine language” to be the processor's internal bit patterns representing instructions and data types. Because it's difficult to know (and work with) these bit patterns, we use assemblers and compilers to convert a program from forms manageable by humans into the forms needed by the processor.

[28] The modern z/Architecture PSW is quite different!

Even though Assembler Language is considered a lower-level language, we rarely program digital computers in “machine language”, so it is no longer accurate to say we program in machine language.[29]

4.8. Processor Evolution

Since the early days of System/360, many updates, changes, enhancements, and improvements have been made to the original architecture. These have included 31-bit and 64-bit addressing (which we'll see in Section 20), 64-bit registers, and a vast variety of new instructions. Many of the instructions we'll see didn't exist in System/360. Each generation of processors has introduced small and large enhancements; while we'll start with basic instructions that have been used for many years, we'll also see many new forms that can simplify programming chores that were more difficult or expensive when only the older instructions were available.

IBM has tried very hard to ensure that existing applications continue to execute correctly on each new generation of processors. This concern with “backward compatibility” has made it easy for users to increase the capacity and performance of their systems without having to rewrite and retest large applications in which they have invested considerable time and effort.

Backward compatibility doesn't apply as uniformly to specialized programs that use system-specific features, but most such features are typically managed by the operating system.

Terms and Definitions

decode
: The CPU action of analyzing the contents of the IR to determine the validity and type of instruction.

exception condition
: A condition indicating an unusual result. Some exceptions can deliver a default result if an interruption has been masked off by appropriate settings, while others always cause an interruption.

execute
: The CPU's action of performing the operation requested by the instruction in the IR.

fetch
: The CPU action of bringing halfwords from memory into the Instruction Register to be interpreted as an instruction.

IC
: Interruption Code, a value indicating the cause of an interruption.

ILC
: See *Instruction Length Code*.

Instruction Length Code
: A 2-bit field in low storage indicating the length in halfwords of an instruction that caused a particular type of interruption.

interruption
: A process taking control away from the currently executing instruction stream, saving information about the interrupted program, and giving control to the Operating System Supervisor.

IR
: Instruction Register, a conceptual internal register in the CPU into which fetched instructions are placed.

---

[29] But some hardy souls still make corrective “patches” to programs in machine language, or enter machine language instructions into memory using various testing and debugging techniques.

machine language
: The internal representations of instructions and data processed by a computer.

operation code
: The portion of an instruction specifying the actions to be performed by the CPU when it executes the instruction. Often called "opcode".

PM
: Program Mask, a 4-bit field in the PSW used to control whether or not certain types of exception conditions should cause an interruption, or take a predefined default action.

5. Memory Addressing

```
 55555555555
555555555555
55
55
555555555
 5555555555
        555
         55
         55
        555
55555555555
555555555
```

We now describe how the CPU calculates addresses of data and instructions in memory when it decodes the instructions of your program.

The addressing technique used in System z differs from that found in many earlier computers, where the *actual* memory address (or addresses) of the operand (or operands) was part of the instruction.

Figure 17. Typical instruction format for old computers

When memory sizes were limited, this was a reasonable and efficient choice.[30]

Because the original System/360 architecture allowed addressing up to 2^{24} bytes of memory, the older technique of placing actual operand addresses into the instructions would have required at least a 24-bit field for each such address. Since few processors had as many as 2^{24} bytes of memory, and because few programs needed as many as 2^{24} bytes of memory to execute, many of the bits in the 24-bit address field would be wasted by such a direct-addressing technique, and instructions would be longer than needed.

In System z, the scheme used for addressing memory operands is much more flexible than using actual operand addresses, and more economical in using the bits allotted to each instruction; but more complex in the way it determines operand addresses.

The System z family of processors supports three *modes* of addressing. This section describes a fundamental type of base-displacement address generation with 24-bit addresses. Section 20 in Chapter VI describes 31-bit and 64-bit addressing, as well as two other types of address generation.

[30] Another reason is that memory was *very* expensive! A really big machine might have had as many as 128 kilobytes of memory; modern processors can have billions of times more.

5.1. The Addressing Halfword

To refer to data or instructions in memory, a program will almost always use one of the general registers, because the CPU uses information in a part of many instructions called an "addressing halfword". An addressing halfword always occupies a halfword in memory.

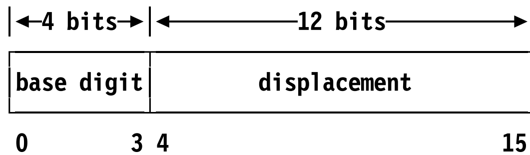


Figure 18. Structure of an addressing halfword

The first 4 bits of the addressing halfword contain a hex digit called the *base register specification digit*, or *base digit*.[31] The base digit specifies a general register called the *base register*. The 12-bit field in the rest of the addressing halfword contains an unsigned nonnegative number called the *displacement* that takes values from 0 to 4095.

To generate the address of an operand, the CPU does the following:

Step 1: The 12-bit displacement is put at the right-hand end of an internal register called the Effective Address Register (abbreviated "EAR"), and the leftmost bits of the EAR are cleared to zeros.

Step 2a: If the base register specification digit is *not* zero, then the contents of the specified general register (the *base register*) are added to the contents of the EAR, and carries out the left end are ignored.

Step 2b: If the base register specification digit is zero, nothing is added to the EAR (so that general register zero will *never* be used by the CPU as a base register). That is, a zero base digit means "no register".

The result in the EAR is called the *Effective Address*. It may be used as the address of an operand in memory, and for many other purposes (such as a shift count). These steps are sketched in Figure 19.

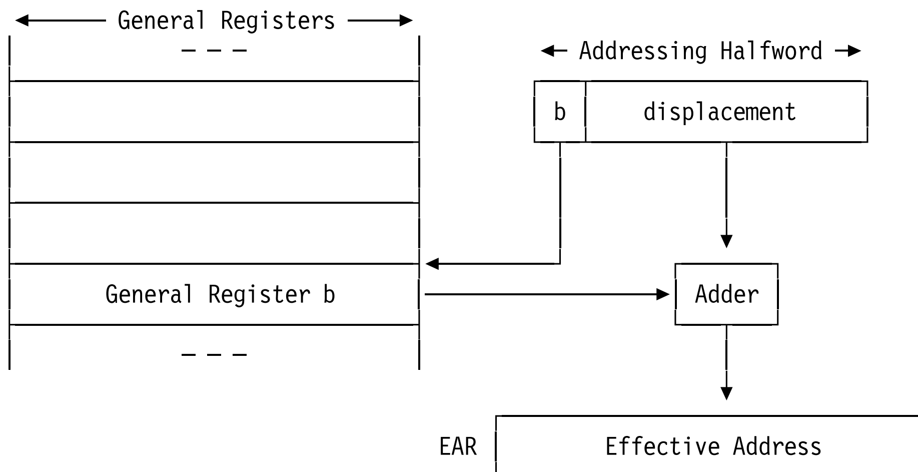


Figure 19. Sketch of Effective Address calculation

This method of generating addresses is called *base-displacement addressing*. In 24-bit addressing mode (which we're assuming for now), only the rightmost 24 bits of the Effective Address are used.

[31] The base register specification digit was sometimes called the "base register address", but this is misleading because the base registers aren't "addressable" like bytes in memory.

Remember

An addressing halfword is not an address. It can be *used* to form an Effective Address.

Exercises

5.1.1.(2) The use of the term "halfword" in describing an addressing halfword implies that it (the addressing halfword) lies on a halfword boundary. Is this true under all circumstances?

5.1.2.(1) How many values may be assumed by the base register specification digit? How many registers may be used by the CPU as base registers?

5.2. Examples of Effective Addresses

In the following examples, additions are done in both binary and hexadecimal arithmetic.

1. Suppose the addressing halfword of an instruction is 1011 001011010101 in binary (X'B2D5') and suppose general register 11 contains

   1100 0111 0011 1110 1001 0000 1010 1111

   in binary (or C73E90AF in hex). Then, assuming we are generating 24-bit addresses, the Effective Address of the instruction is

   ```
    0000 0000 0000 0010 1101 0101   0002D5 (displacement)
   +0011 1110 1001 0000 1010 1111   3E90AF (base)
    0011 1110 1001 0011 1000 0100   3E9384 (Effective Address)
   ```

2. Suppose the addressing halfword of the same instruction is X'0468'. Then the Effective Address is X'000468', since general register zero is never used as a base register.
3. Suppose the addressing halfword of the same instruction is X'B000', and the contents of R11 are as before. Then the Effective Address is X'3E90AF'; a zero displacement is valid.

Exercises

5.2.1.(2)+ Assume general registers 0, 1, and 2 contain these values:

```
c(GR0) = X'12001038'
c(GR1) = X'0902A020'
c(GR2) = X'001AAEA4'
```

Calculate the 24-bit Effective Address for these addressing halfwords: (1) X'206C', (2) X'1EEC', (3) X'0FB0'.

5.2.2.(2)+ Assuming the same register contents as in Exercise 5.2.1, calculate the 24-bit Effective Address for these addressing halfwords: (1) X'1FEF', (2) X'0FC8', (3) X'2EA4'.

5.3. Indexing

After the displacement has been added to the base (if any), the CPU again checks the type of the instruction. If the instruction is type RX, an indexing cycle is needed. The second byte of an RX-type instruction (the "register specification" in Table 7 on page 52) contains two four-bit fields: the second is called the *index register specification digit* or *index register digit* or *index digit*, as shown in Figure 20 on page 64.

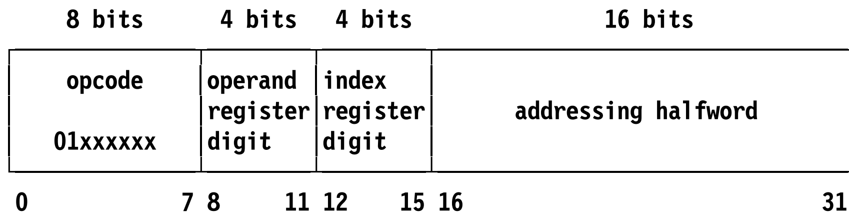


Figure 20. RX-type instruction, showing index register specification digit

Step 3: If the instruction is type RX, *and* the 4-bit index register specification digit is *not* zero, then the contents of the general register specified by the index register specification digit are added to the contents of the EAR (again ignoring carries out the left end). A zero index digit means "no register", not general register zero.

The resulting quantity in the EAR is still called the Effective Address (sometimes called the *Indexed Effective Address*). These steps are sketched in Figure 21.

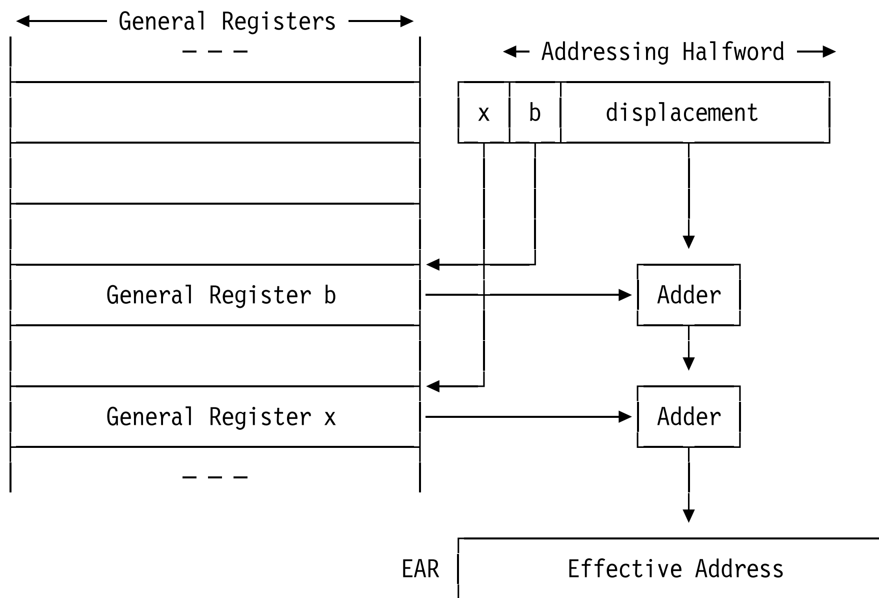


Figure 21. Sketch of Effective Address calculation with indexing

Modern CPUs add the base and index register contents with a three-input adder, so there is actually only one calculation. The index register specification digit is sometimes called the *index digit*; similarly, the specified register is the *index register*, and the quantity in it is the *index*.

Indexing is a powerful way to process structures of data items like arrays with uniform and regular spacing, as we will see in Section 40. The addressing halfword provides the address of a fixed position, and the index selects a particular item.

Exercises

5.3.1.(1) Draw a picture showing the locations of the base register specification digit, the base register, and the base address. Then do the same for the corresponding index quantities.

5.3.2.(1) How does the CPU determine that an indexing cycle is needed during address computation?

5.3.3.(2) For each instruction type, determine the maximum number of general registers that might be accessed by the CPU in calculating Effective Addresses.

5.3.4.(2) Under what circumstances will the CPU not calculate an Effective Address?

5.4. Examples of Indexing

Continuing the examples of calculating Effective Addresss that we saw in Section 5.2:

4. Suppose an RX-type instruction is `X'430A7468'` and that GR7 contains `X'12345678'` and GR10 contains `X'FEDCBA98'`. (The base register specification digit `X'7'` means that GR7 is used as the source of the base address.) Again assuming we are generating 24-bit addresses, the Effective Address is

```
 0000 0000 0000 0100 0110 1000   000468 (displacement)
+0011 0100 0101 0110 0111 1000   345678 (base, from GR7)
 0011 0100 0101 1010 1110 0000   345AE0
+1101 1100 1011 1010 1001 1000   DCBA98 (index, from GR10)
 0001 0001 0001 0101 0111 1000   111578 (Effective Address)
```

5. Suppose an RX-type instruction is `X'43007468'` and that the contents of GR7 are again `X'12345678'`. Then the Effective Address is

```
 0000 0000 0000 0100 0110 1000   000468 (displacement)
+0011 0100 0101 0110 0111 1000   345678 (base)
 0011 0100 0101 1010 1110 0000   345AE0 (Effective Address)
```

   (No indexing cycle is needed, since the index register specification digit is zero.)

6. Suppose an RX-type instruction is `X'43070468'` and that GR7 still contains `X'12345678'`. Then the Effective Address is

```
 0000 0000 0000 0100 0110 1000   000468 (displacement)
+0000 0000 0000 0000 0000 0000   000000 (base)
 0000 0000 0000 0100 0110 1000   000468
+0011 0100 0101 0110 0111 1000   345678 (index)
 0011 0100 0101 1010 1110 0000   345AE0 (Effective Address)
```

   In this example the values of the base and index register specification digits were interchanged from those in example 5, so that the indexing cycle was required to compute the same Effective Address.

In situations where only one register is used to calculate an Effective Address (as above, where the base digit was 0 and the index digit was 7), be careful not to call that register the *base* register, even though it usually behaves like a base register in an RX-type instruction.[32]

Exercises

5.4.1.(1) Under what circumstances may GR0 be used as a base register? As an index register?

5.4.2.(3)+ Assume the hexadecimal contents of the general registers are as shown:

```
C(GR0) = 12001028   C(GR4) = 8888000E
C(GR1) = 8902A020   C(GR5) = 12345678
C(GR2) = 4F1AAEA4   C(GR6) = 0FDE3B72
C(GR3) = FFFFFFF8   C(GR7) = 92837465
```

and GR8 through GR15 contain zeros. Now, compute the 24-bit Effective Address of each of the following instructions, paying careful attention to instruction type: (1) `X'9803206C'`, (2) `X'50F10EEC'`, (3) `X'41133333'`, (4) `X'7A341DA4'`, (5) `X'91220166'`, (6) `X'8F120FB0'`.

5.4.3.(3)+ Assume that the contents of the general registers are as shown below for GR0 through GR7, and that GR8 through GR15 contain zeros.

[32] In the “Access Register” addressing mode, index and base registers participate differently in calculating Effective Addresses: only base registers are used to select an Access Register.

```
C(GR0) = 00000044   C(GR4) = 41800000
C(GR1) = 000902AE   C(GR5) = 00010000
C(GR2) = A20710FC   C(GR6) = 00FFFF00
C(GR3) = FFFFFFFF   C(GR7) = FF000000
```

Now, compute the 24-bit Effective Address of each of the following instructions: (1) X'41726100', (2) X'920710FC', (3) X'7A333002', (4) X'5806016C', (5) X'43B00044', (6) X'90EC126A', (7) X'86052E4D'.

5.4.4.(3) Suppose the contents of the general registers are as shown in Exercise 5.5.2 below. For each of the following instructions, determine the Effective Address, paying careful attention to instruction type: (1) X'58040404', (2) X'91628DBC', (3) X'44FF7D5C'.

5.5. Addressing Problems (*)

The Effective Address in the EAR has many uses, most often to address operands in memory; it is also used for other purposes such as shifting and branching.

Certain instructions operating on groups of bytes require the address of the leftmost (lowest-addressed) byte of the operand group be *exactly* divisible by the length of the operand. If this condition is not satisfied, a program interruption for a *specification exception* occurs. In early processors, operand alignment was required for almost all instructions, but the requirement was relaxed soon after.[33] Few instructions in modern processors require strict operand alignment.

When you use base-displacement addressing with 12-bit displacements, the only part of the memory that can be referenced without using a base register is the area with addresses 0 to 4095 = X'FFF', so you will almost always use a base register to refer to operands in memory. (We'll see in Chapter VI, Section 20 that instructions with signed 20-bit displacements make this 4K-byte limitation much less severe.)

You can't put your program into those first 4096 bytes[34] because that area of memory (and more) is reserved by the CPU and the Operating System. This means that if you want to access a byte in memory at address XX (where XX is greater than 4095), there must be a base register available — one of registers 1 to 15. If a base register contains a *base address*, and XX lies between that base address and the base address+4095, then we say that XX is *addressable*. If there is no such number in a register, then the byte at XX is *not addressable* by your program.

When we place a number in a register to address a 4096-byte region of memory, that register provides *addressability* for the region. However, if the number itself must be brought from another portion of memory that is not currently addressable, we are back where we started, needing another number to provide addressability for the first number.

Fortunately, there are simple solutions to the problems of establishing addressability. The BASR instruction is often used (as we will see soon), and the Assembler's *address constants* also allow us to refer to other areas of our program. Modern processors provide new ways to minimize these addressing problems: *long displacements* and *relative addressing*. We will turn to them in Section 20 after we have investigated the most often-used instructions.

Exercises

5.5.1.(3)+ Suppose the general registers contain the values shown in Exercise 5.2.1. Which of the following locations in memory (given in hexadecimal) are addressable through the use of the base-displacement addressing technique? For each location that *is* addressable, derive an addressing halfword that can be used to address it. (1) X'02ABCD', (2) X'000A4D', (3) X'001139', (4) X'88888E', (5) X'02A010'.

---

33 Because many programs had to manage unaligned data items, extra instructions were needed to isolate and align the required item. The processor designers were asked (urgently!) to remove the restriction wherever possible. The relaxation of the alignment requirement was called the “Byte-Oriented Operand Feature”; it soon was known as the “BOOF”.

34 Unless you're writing your own operating system!

5.5.2.(3)+ Suppose the contents of the general registers are as follows:

```
C(GR0) = 00010A20       C(GR8) = 8031B244
C(GR1) = 42319B7C       C(GR9) = 00000010
C(GR2) = 91F0F002      C(GR10) = 723B94C1
C(GR3) = 1002340A      C(GR11) = E931AB7F
C(GR4) = 00FF00FF      C(GR12) = 00000E38
C(GR5) = D907C401      C(GR13) = 6B005000
C(GR6) = 12345678      C(GR14) = 80000000
C(GR7) = 992B42A3      C(GR15) = FFFFFFFF
```

For each of the following memory addresses, determine first whether or not that memory location is addressable by a program using those registers. If it *is* addressable, determine an addressing halfword (base-displacement halfword) that can be used to address the location. (1) X'010A20', (2) X'FFFFFF', (3) X'6A0054', (4) X'31AB7E', (5) X'001234', (6) X'07D3C4', (7) X'00A004', (8) X'31BB65', (9) X'9ABCDE', (10) X'07C401'.

5.5.3.(3) In Exercise 5.5.2, which locations are addressable through the base-displacement addressing technique *with* indexing allowed? Derive an addressing halfword and the accompanying index digit that (in an RX-type instruction) would make the locations addressable.

5.5.4.(3)+ Suppose the contents of the general registers are as shown in Exercise 5.1.2 on page 63 (note that registers 8 through 15 contain zeros). For each of the following memory addresses, determine an addressing halfword that can be used to address that memory position. If no such addressing halfword exists, say so. (1) X'000EEB', (2) X'001040', (3) X'072000'. How many solutions are there for address (1)?

5.5.5.(4)+ In Exercise 5.5.1, which locations in memory are addressable through the base-displacement addressing technique *with* indexing allowed? Derive an addressing halfword and the accompanying index digit that (in an RX-type instruction) would make the locations addressable. (Remember that Exercise 5.5.1 refers to Exercise 5.2.1.)

5.5.6.(1) Suppose a program can be put entirely within the first 4096 bytes of memory. Will it use GR0 as a base register?

5.5.7.(2) Assume that the contents of the general registers are as shown in Exercise 5.5.2. For each of the following SS-type instructions, compute *both* Effective Addresses (there are two addressing halfwords in an SS instruction, as shown in Table 9 on page 53).
(1) X'D2078F1D57C4', (2) X'DCFFDCFF7000', (3) X'F26337390050', (4) X'D58DFE4FC016'.

5.6. Address Translation and Virtual Memory (*)

All models of System z support address translation, called *Dynamic Address Translation* (or "DAT"). Address translation is invisible to application programs. It provides greater Operating System flexibility in assigning programs to main memory, a heavily used resource. Address translation takes your program's "virtual" addresses and maps them invisibly into the "real addresses" needed for references to "real" memory.

Without DAT, a reference to a byte at X'123456' addresses that byte in the physical or "real" memory of the processor. When DAT is active, your reference to a byte (at *your* virtual Effective Address X'123456') is translated into a "real" address (such as X'27D94FA') having no obvious relation to your address; you can't determine the relation of your virtual addresses to the real addresses to which they are mapped. The Operating System, working with the DAT facilities, makes it possible for your program to operate as though it is addressing "real" memory; but only the Operating System works with real addresses. This is why your addresses are called "virtual" — they aren't real.

Address translation is simple in concept but complex in implementation. To illustrate, the virtual (effective) address supplied by your program is divided into sections; for 31-bit addresses, they are a *segment index*, a *page index*, and a *byte index*, as illustrated in Figure 22 on page 68.

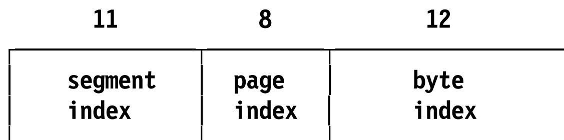


Figure 22. 31-bit Virtual Address

To use these indexes for calculating real addresses, the Operating System first constructs (in a protected area of real memory) two sets of tables, *page tables* and a *segment table*, and it places the address of the segment table (for example, taken from Control Register 1) into an internal field. Your virtual address is translated into a real address roughly as follows:

Step 1: The segment table address is retrieved and the segment index is added to it. The result is the address of one of the entries in a list of segment tables.

Step 2: The specified segment table entry (which contains the address of one of the entries in a list of page tables) is retrieved, and the page table index is added to it. The result is the address of an entry in the specified page table.

Step 3: The specified entry in the page table is retrieved, and attached to the left (high-order) end of the byte index. The result is the *real address* of a byte in main memory.

We will not show examples of translation, since it is invisible to your program.

This description covers only very basic aspects of translation, and does not cover 64-bit virtual addresses. There are many other details of the process, and (because translation is very heavily used) the processor has a lot of additional hardware to optimize the process.

Exercises

5.6.1.(3) Some processors use a technique called *indirect addressing*. If a bit in the instruction (called the *indirect-addressing* bit) is nonzero, the CPU uses the Effective Address not to access an operand, but to access a second instruction. The Effective Address of this new instruction then becomes the operand address that points to the desired operand. (On some processors, if the instruction at the “indirect address” had *its* indirect-addressing bit set, then the entire process repeats until an instruction is found without the indirect-addressing bit set.) Can you think of reasons why indirect addressing is not provided by System z?

5.6.2.(0) Another aspect of early addressing techniques (whereby instructions contained actual operand addresses) was that the address portions of instructions often had to be modified. Find a programming “old-timer”: ask for an explanation of address modification techniques on processors such as the IBM 7090, and why the method used on System z is so clearly superior.

5.7. Summary

As noted earlier, Effective Addresses are used for many purposes; the most common is to refer to an operand in memory. Almost always, the operand is referred to by its lowest-addressed byte; and if the operand is a binary integer, that byte contains the most significant (*high-order*) byte of the integer. So, references to “low-order” and “high-order” may need to distinguish clearly between memory addresses, bit ordering, and numeric significance.

Terms and Definitions

address translation (“Dynamic Address Translation”, DAT)
The procedure used by the CPU to convert virtual addresses into real addresses.

addressability
A base register and a displacement provide an Effective Address allowing valid reference to a byte in memory.

addressing halfword
: A halfword containing a *base register specification digit* in the first 4 bits, and an unsigned *displacement* in the remaining 12 bits. A key element of System z addressing.

base address
: The execution-time contents of a base register.

base register
: A general register used at execution time to form an Effective Address.

base register specification digit
: The first 4 bits of an addressing halfword.

displacement
: An unsigned 12-bit integer field in an addressing halfword used in generating Effective Addresses.[35]

EAR (Effective Address Register)
: A (conceptual) internal register used to hold Effective Addresses.

Effective Address
: The address calculated from an addressing halfword, possibly with indexing.

index
: The contents of an index register.

index register specification digit
: 4 bits of an RX-type instruction specifying a register with a value to be added to the Effective Address calculated from an addressing halfword.

indexing
: Computation of an Effective Address by adding a displacement to the contents of a base register and an index register.

real address
: The "true" address of a memory location.

virtual address
: The address of a memory location that may physically reside at a different real address.

[35] We will see in Section 20 that System z provides another form of base-displacement addressing with a signed 20-bit displacement.

Chapter III: Assembler Language Programs

```
IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
    II          II          II
    II          II          II
    II          II          II
    II          II          II
    II          II          II
    II          II          II
    II          II          II
    II          II          II
IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
```

We have seen how the CPU executes instructions and evaluates addresses; now we'll see how we write Assembler Language programs.

- Section 6 describes typical steps involved in preparing, assembling, linking, and executing programs written in Assembler Language.
- Sections 7 and 8 examine the components from which machine, assembler, and macro instruction statements are formed.
- Section 9 describes five major machine-instruction types and how we write their operands in machine instruction statements.
- Section 10 introduces the key concept of *addressability* in Assembler Language programs, a necessary step for any program executed on System z.

6. Assembler Language

```
 6666666666
666666666666
66        66
66
66
66666666666
666666666666
66        66
66        66
66        66
666666666666
 6666666666
```

The Assembler is the program most used in creating specific instruction sequences for execution by the processor.

First, we describe how to write programs and see the steps leading to their execution. The conventions and rules for using the Assembler are called "Assembler Language", even though there is little resemblance to what most people mean by "language".

6.1. Processing Your Program

First, we consider the steps involved in running an Assembler Language program:

1. assembly
2. linking
3. loading and execution

6.1.1. Assembly

Assembly is represented schematically in Figure 23. The Supervisor places the Assembler in memory to begin assembling your source program.

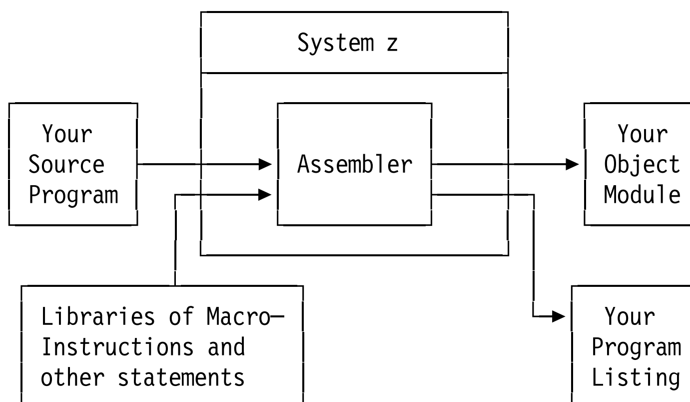


Figure 23. Simple view of Assembler processing

The Assembler reads the *statements* of your Assembler Language program, processes them — possibly with the help of some data in libraries of macro-instructions and other statements — converts your Assembler Language program to machine language, and produces an *object module* containing *object code*. Usually you will want a program listing showing your source program and the generated *object code*, with additional information about the Assembler's processing and indications of errors it may have detected.

The Assembler converts the program from a form convenient for you (statements) to a form convenient for the processor (binary data and instructions), its *machine language.*

6.1.2. Linking

The Linker[36] combines your object module with any others needed for execution. The linking step is sketched in Figure 24; the Linker is placed in memory and begins execution.

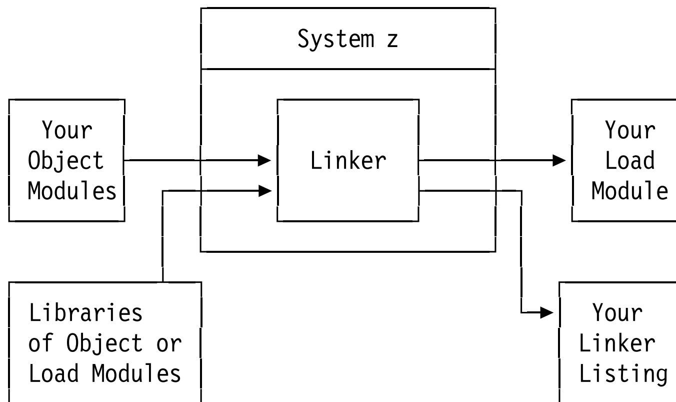


Figure 24. Simple view of program linking

The output of the Linker is a *load module*.[37] The load module is written to a storage device, and a listing of information summarizing the linking process is created.

The Linker also accepts load modules as input, allowing you to update or modify existing load modules without having to reassemble all its components.

6.1.3. Loading and Execution

At execution time, the load module produced in the linking step is "loaded" into memory. An essential feature of this process is *relocation*, which we'll investigate in Chapter X, Section 38. The portion of the Supervisor that loads and relocates load modules is called the *Program Loader*. Like the Linker, it is a program that treats other programs as data.

After your program has been loaded into memory, the Supervisor transfers control to it by setting the Instruction Address in the PSW to the address of the instruction where you want execution to begin. Your program then does whatever processing you told it to do[38] and when it is finished it returns control to the Supervisor.

[36] We'll use "Linker" to mean any program (such as the Binder and Linkage Editor) that combines object module files into executable files like load modules.

[37] The output of a Linker has many many different names and forms, depending on the operating system and the system Linker. For example, on System z the output of the z/OS binder can be a "load module" or a "program object"; the output of the z/VSE Linker is a "phase", and the output of the z/VM CMS loader is a "module". We'll use "load module" to mean a data set or file ready to be loaded directly into memory for execution.

[38] Which may not always be what you intended!

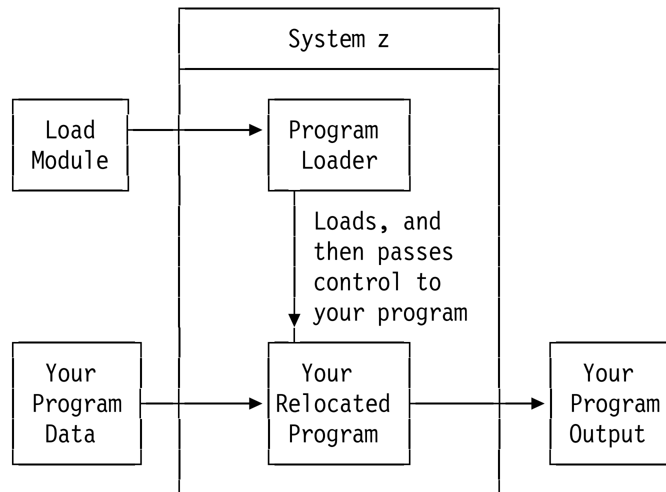


Figure 25. Simple view of program loading and execution

The last two linking and program-loading steps can be combined by using a *Loader* instead of the Linker and Program Fetch routines. The Linker or Loader reads and relocates your object modules directly into memory, and combines them with any necessary additional object and load modules from the "Libraries of Object or Load Modules".

An Assembler Language program is "processed" *twice*: once by the Assembler at *assembly time*, and once by the CPU when it is executed at *execution time* (or *run time*). The difference between these two times is important: the Assembler produces object modules with machine language instructions and data to be placed into memory later; your data is processed only when your program is finally loaded and your instructions are executed.

Exercises

6.1.1.(1) Draw a diagram combining Figures 23 through 25, to show the relationships between the inputs and outputs of processing your programs at each step.

6.2. Preparing Assembler Language Statements

You prepare Assembler Language programs in the form of *statements*. There are four types: *comment* statements, *machine instruction* statements, *assembler instruction statements*, and *macro-instruction* statements. All four can be used in creating programs.

1. Comment statements provide explanatory material in the program so it will be easier for you and others to read and understand. They are displayed in the program listing, but are not translated into instructions or data and do not appear in the object module.
2. Machine instruction statements are converted by the Assembler into machine language instructions for the CPU to execute when your program is loaded into memory for execution.
3. Assembler instruction statements provide information to the Assembler. They can be as simple as statements generating data or specifying a title for the top of each page of the listing, or can be more complicated, such as statements telling the Assembler that certain registers may be used as base registers. Some Assembler instruction statements cause the Assembler to generate machine language data; others do not.
4. Macro instructions provide a compact assembly-time notation for groups of statements. They are a convenient way to specify sequences of other statements (all four types are allowed) in which parts of the generated statements can be changed to suit your needs. Macro instructions are a very powerful and useful feature of the Assembler Language.

The Assembler processes input records exactly 80 bytes long. Your records may not extend all the way to 80 characters, but there must still be enough blank or other characters to extend its length to 80. These 80-character records are often called "card-image" records.[39]

Statements occupy positions 1 through 71 of a line. Such positions are called "columns". Column 72 has a special meaning: if it is *not* blank, the *next* line is considered to be a continuation of the line with the nonblank character in column 72, in such a way that the character in column 16 of the second line is treated as following immediately after the character in column 71 of the preceding continued line.[40] This is illustrated in Figure 26. These conventions — column 72 for the continuation indicator and column 16 where the statement continues — are almost always used for machine instruction and assembler instruction statements.

Columns 73-80 may be used for any purposes (usually, for sequencing data).

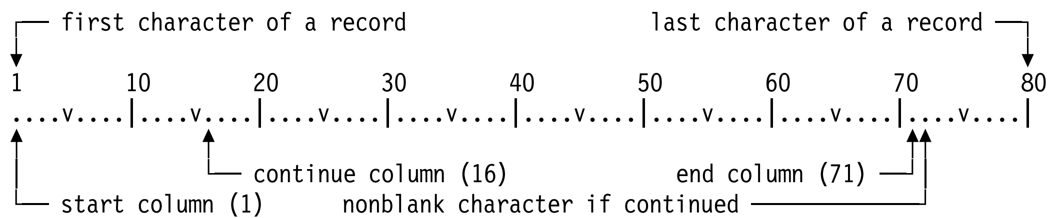


Figure 26. Assembler Language statement columns

Columns 1 through 15 of a continuation line must be blank. (A common error is to write characters in column 72 accidentally, so that the following line is treated as a continuation line, and processed in an unexpected way.)

Columns 73 through 80 are ignored by the Assembler. Since all 80 columns of the input record appear on the listing, the last 8 columns are often used for identification or sequencing information.[41]

A comment statement is identified by an asterisk (*) in column 1. Any information may appear in columns 2 through 71. Figure 27 on page 76 has examples of comment statements:

[39] The choice of 80 characters goes back to the nearly universal use of "IBM cards". For many years before and after the introduction of System/360, programs and data were prepared on 80-column punched cards. So, we still say "column" rather than something like "position".

[40] You can change these columns with the ICTL Assembler instruction statement. It allows other columns to be used for the start, end, and continuation columns of a statement. The numbers given are the ones the Assembler uses if it is not told otherwise. ICTL is almost never used, anyway; if you use ICTL to change those columns, other readers of your program may be confused.

[41] Even though IBM cards have 80 columns, early computers like the IBM 704 and 709 couldn't read the last 8 columns! Those processors had 36-bit words, so their card readers read alternate groups of 36 bits from the 12 rows on a card into 24 words. This 72-column custom persists.

```
1          10          20          30          40          50          60          70          80
....v....|....v....|....v....|....v....|....v....|....v....|....v....|....v....|

 *  This is a comment statement. It is not continued.

 *  This comment statement is correctly continued: its continuation     X ←column 72
               on this next line starts in column 16.

 *  This comment statement is also continued, but is an error:          X
    this continuation line has nonblank characters before column 16.
```

Figure 27. Comment statement examples

Figure 27 contains some entirely blank lines. They are often used to improve readability; the Assembler copies them to the program listing, and they have no effect on your program.

Comment statements may be continued onto following lines, as shown in the figure above. This is generally not a good practice; most programmers avoid column 72 in comment statements.

A common method for adding "blocks" of comments to a program is illustrated in Figure 28.

```
*********************************************************************
*
*  This is a block of comments documenting the behavior of this
*  program. Since we have not written any programs yet, this block
*  only illustrates how you can include large amounts of descriptive
*  text to your program to help readers and maintainers understand
*  what the program does -- at least, what you intended it to do.
*
*********************************************************************
```

Figure 28. Block comments

Exercises

6.2.1.(1) For the Assembler you use, determine what rules apply to the columns of continued statements after the first continuation.

6.3. Statement Fields

The machine instruction, Assembler instruction, and macro-instruction statements each have four parts called *fields*: the *name*, *operation*, *operand*, and *comment* or *remarks* fields.[42] An entry in the operation field must always be present, and for certain statements an entry in some of the other fields may or must be omitted.

If there is a name field entry in the statement, it *must* begin with a nonblank character in column 1. It is terminated by the first blank column after column 1. If no name field entry is desired, column 1 must be left blank.

[42] It's better to call this the "remarks" field, to avoid confusion with comment *statements*.

After the name field and separated from it by one or more blanks is the operation field entry; it ends with the first blank after the start of the operation field. The operation field entry is sometimes called the “mnemonic” or “operation” or “operation mnemonic”.[43]

After the operation field entry and separated from it by one or more blanks is the operand field entry, which, like the name and operation field entries, terminates with the first blank column detected after the start of the operand field entry, except for one special case (quoted strings) described in the next section.

The rest of the input line is treated as remarks by the Assembler and is ignored. It does not influence the processing of the statement *unless* this field extends into column 72, indicating a continuation on the next line. Except for the name field, there is no restriction on the columns where the other three fields must start; they simply end with a blank column.

This allows *free-field* statements: you can arrange the information on the input lines of your program as you like, but the fields must appear in the proper order. These rules are summarized in Figure 29, where “⊥” means “one or more blanks”.

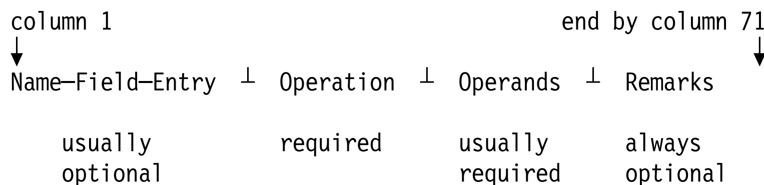


Figure 29. Statement fields for machine, assembler, and macro-instruction statements

Even though any number of blanks can be used to separate the fields of a statement, it is customary to improve program readability by making all operation, operand, and remarks fields entries start in the same columns. For example, if your name-field entries are eight or fewer characters long, place your operation field entries in column 10; similarly, if the operation field entries are eight or fewer characters long, start your operand field entries in column 19. Later examples of program fragments will show how this can be done.

A good programming practice is to use the remarks field to tell the reader what the statement is *supposed* to do, and *why*. (Program comments and remarks sometimes say the program “does” one thing, while it actually does something different when the CPU executes it!)

Good Programming Practice

Your program's comments and remarks should help the reader (who may be you!) understand what each statement and group of statements is doing, and why.

The term “operand” can be confusing. Section 3.1 on page 43 stated that an operand is something in a register or in memory that is “operated on” during the execution portion of the instruction cycle. “Operand” is *also* used here to describe the components that make up the operand field entry of a statement! It helps to remember that the first meaning applies to the execution step of a job, while the second meaning applies only during the assembly step.

Figure 30 on page 78 illustrates a *machine instruction statement* in which entries appear in all four fields.

[43] Be careful not to call it the “opcode”! That term is properly used for the bits of an instruction that tell the CPU what to do. Sometimes people use “opcode” to mean both the operation field entry of an instruction — the mnemonic — *and* the machine instruction bits, so listen carefully. Which is meant will usually be clear.

```
LOAD     LR    7,3          Copy c(GR3) to GR7
```

Figure 30. A machine instruction statement

The operand field entry has two entries, “7” and “3”, separated by a comma. If the instruction is executed in a program, it would cause the contents of general register 7 to be replaced by a copy of the contents of general register 3.[44]

The *assembler instruction statement* in Figure 31 omits the name and comment field entries, and causes the Assembler to put a “title” heading on each page of the program listing.

```
         TITLE 'PROGRAM NO. 1'
```

Figure 31. An assembler instruction statement

Figure 32 shows an example of a *macro-instruction statement* in which only an operation field entry appears.

```
         RETURN
```

Figure 32. The macro-instruction statement RETURN

If the RETURN statement above had been prepared in the days of punched cards, the card[45] might look like this:

```
         RETURN
          █
         █   ██
00000000000██0000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333█33333333333333333333333333333333333333333333333333333333333333333333
444444444444█4444444444444444444444444444444444444444444444444444444444444444444
5555555555█555█55555555555555555555555555555555555555555555555555555555555555555
66666666666666666666666666666666666666666666666666666666666666666666666666666666
77777777777777777777777777777777777777777777777777777777777777777777777777777777
88888888888888888888888888888888888888888888888888888888888888888888888888888888
999999999█999█99999999999999999999999999999999999999999999999999999999999IBM5081
```

Table 12. Punched-card image of a RETURN statement

The Assembler supports mixed-case characters, so you need not write symbols, operation field entries, and most operand field entries using upper-case letters. (However, the Assembler treats lower-case and upper-case letters as equivalent when they appear in symbols and operation field entries; unlike some high-level languages, the Assembler is not case-sensitive except for characters within quoted strings.) Thus, you could write Figure 32 as

44 The remarks in this statement are quite useless, because readers can see what the instruction does. Remarks should explain reasons for doing something, like “`Copy record count to GR7 for multiplication`”.

45 The characters “IBM5081” in the bottom right corner of the “card” were called the “electro number”, the number of the plate used for printing the cards. Number 5081 was used for cards with no other information than the row numbers, zero through nine. The empty two rows at the top were called the “twelve” row and the “eleven” row. (This card was also known as the “IBM Model 5081 Data Storage Device”.)

```
Return
```

with the same results.

> **Mixed Case Names: Be Careful!**
>
> The Assembler accepts mixed-case names, but processes them internally as through they are all in upper case. Thus, a symbol like **AbCdEfgh** is considered to be the same as the symbol **ABCDEFGH**.

6.3.1. What's in a Name Field? (*)

Many items can appear in the name field of an instruction statement, such as:

- the name of a machine instruction
- the name of a data area
- a symbol to be given a value without naming any part of the program
- a Labeled USING qualifier (described in Chapter XI, Section 39.4)
- in the Conditional Assembler Language, a variable or sequence symbol
- characters to be copied to the sequence field of the object module
- ... and some statement require the name field to be empty!

Some people call the name field entry a “label” when it is the name of a machine instruction, but in other contexts this can be very misleading. It's too easy to start thinking of all name-field symbols as “labels” when they're actually used for other purposes.

Exercises

6.3.1.(1) Suppose a program contains the machine instruction statement shown in Figure 30 on page 78. During what part of the job processing will the statement be read by the Assembler? During what part of the job processing will the assembled hexadecimal instruction be fetched by the CPU?

6.3.2.(1) In what column should the remarks field of a machine instruction statement begin?

6.3.3.(1) In what columns may the operation field entry of a machine instruction statement begin?

6.3.4.(1) Which field in an assembler instruction statement is required?

6.3.5.(2)+ What types of statement may be written without an operation field? Without an operand field? Without a remarks field?

6.3.6.(2)+ Suppose the machine instruction statement in Figure 30 on page 78 had been written so that column 1 was blank, and the characters “LOAD” began in column 2. How would the fields of the statement be interpreted?

6.3.7.(2) What types of Assembler Language statements may be written without an operation field? Without a comment field?

6.4. Writing Programs

While these basic rules are nearly complete, you will be able to write executable programs after we cover a few necessary details.

A program is a sequence of Assembler Language statements. The input to the Assembler should consist of

1. a START statement,
2. the statements of your program, and
3. an END statement.

The START statement is written

```
progname START origin
```

The name-field symbol **progname** is the name of the program. It will usually have eight or fewer letters. The **origin** operand is called the *initial location* or *assumed origin* of the program; its value is used by the Assembler. For now, we will use zero for this initial location. Thus, the *first* statement of a program should be something like

```
TEST     START 0                 First statement of program TEST
```

where **TEST** is the name of your program.

The *last* statement of the program must be an END statement telling the Assembler to stop reading records. It is written

```
         END   progname          Last statement of program
```

where the **progname** operand of the END statement should (for now) be the same as the **progname** in the name field of the START statement. For our example, we would write

```
         END   TEST              Begin execution at 'TEST'
```

The **progname** in the operand field of the END statement specifies the name of the instruction where execution should start when the program is loaded into memory. The operand field entry on the END statement may be omitted, but specifying it is a good programming practice, so we'll write our sample programs this way.

The Assembler allows *no* symbol as the name-field entry in an END statement. Assembler Language programs, unlike programs in many high-level languages, must not try to terminate execution by allowing control to reach the END statement. Doing so usually results in some form of disaster, since the END assembler instruction statement only tells the Assembler to stop reading records, and is not translated into executable instructions.

The START and END statements, when read by the Assembler, determine the beginning and end of the statements to be assembled. The START statement may be preceded by a few types of statements (such as TITLE and comment statements), but for now, assume it is the first statement to be read. The END statement may not be followed by *any* other statement: it must be last.

Some programmers like to start their programs with a CSECT ("Control SECTion") statement rather than START. It has the same effect, except that no operand field entry is allowed, so you can't set the initial location or assumed origin value. We'll discuss control sections and the CSECT instruction thoroughly in Chapter X, Section 38.

6.5. A Sample Program

Figure 33 on page 81 is a little program that prints my name. This set of records is typical of those required on many System z systems. All statements begin in column 1 and end before column 72. (The "Line n" comments are used only for this example; you don't need them for your programs.)

```
◄──────────────────────────── 80 characters ────────────────────────────►
//JRETEST JOB (A925,2236067977),'J.EHRMAN'                          Line 1
//     EXEC ASMACLG                                                 Line 2
//C.SYSIN   DD *                                                    Line 3
Test   Start 0           First line of program                      Line 4
       Print NoGen                                                  Line 5
*      Sample Program                                               Line 6
       BASR  15,0        Establish a base register                  Line 7
       Using *,15        Inform the Assembler                       Line 8
       PRINTOUT  MyName,*  Print name and stop                      Line 9
MyName DC    C'John R. Ehrman'    Define constant with name         Line 10
       END   Test        Last statement                             Line 11
/*                                                                  Line 12
```

Figure 33. A complete Assembler Language program

The first 3 lines and the last are *control statements* for the Supervisor; they are not part of your program, and are not read by the Assembler. They tell the operating system to run the Assembler, Linker, and Program Loader, and how to pass your program's statements to the Assembler. The information on these lines follows the rules of a *Job Control Language* for an operating system. Line 1 (the JOB statement) marks the beginning of a *job*: a unit of work for the computer separate from all other units. Additional information on the JOB statement provides accounting data such as an account number and a user name.

Line 2 (the EXEC statement) requests that the following program be assembled, linked, and executed; Line 3 indicates that records for the Assembler follow immediately. The last line (the "**/***" or "end-of-file" statement) tells the Operating System that no more records are given to the Assembler.

The Assembler Language program is contained in the remaining lines:

- Line 4 is the assembler instruction statement defining the name of your program as **Test** and starts a Control Section to contain the machine language data and instructions of your program when it is executed by the CPU.
- Line 5 is an assembler instruction statement; it causes the Assembler not to print statements generated by the **PRINTOUT** macro-instruction in Line 9. (More about **PRINTOUT** and other useful macro-instructions in Section 6.6 on page 82.)
- Line 6 is a comment statement.
- Line 7 is a machine instruction statement.
- Line 8 is an assembler instruction statement. (Lines 7 and 8 are important: we'll discuss them in Section 10.)
- Line 9 is a macro-instruction statement that causes some data to be printed, and then returns control to the Supervisor.
- Line 10 is an assembler instruction statement. The Assembler converts the characters enclosed in the apostrophes into an internal form representing the characters.
- Line 11 is an assembler instruction statement. It tells the Assembler that no further statements will be processed for this program. The operand field entry **Test** tells the Linker where you want your program to begin execution.

Exercises

6.5.1.(1)+ Determine what control statements are required at your installation for the following sequences of steps (if they are available): (1) assembling a program, (2) assembling and linking a program, (3) assembling, linking, and executing a program, (4) assembling, loading, and executing a program, and (5) linking and executing an object module created in a previous assembly.

6.5.2.(1) At execution time, if control reaches the END statement, will that be the end of the program?

6.5.3.(1) Examine the Assembler Language program in Figure 33. Which statements have entries in the name field? In the operation field? In the operand field? In the comment field?

6.6. Basic Macro Instructions

For our sample programs, we need only very simple methods of reading 80-character "card-image" records, printing strings of characters, displaying useful information, and displaying or "dumping" areas of memory in hexadecimal format.

Your operating system may provide similar facilities already, but you should check to see how or whether they differ from these. We will use these six macro-instructions, and show how they're used in some programming examples.

PRINTOUT Print formatted information about data and registers

READCARD Read 80-byte card-image records

PRINTLIN Print lines of characters

DUMPOUT Dump memory in hexadecimal format

CONVERTI Convert decimal characters to a 32- or 64-bit binary integer

CONVERTO Convert a 32- or 64-bit binary number to decimal characters

The macro instructions and their operands are described in "Appendix B: Simple I/O Macros" on page 1015.

6.7. Summary

The Assembler provides many facilities to simplify programming tasks.

1. It automatically resolves addresses into the base-displacement and other forms used by System z. The Assembler determines the needed base and displacement so that correct Effective Addresses will be calculated at execution time.
2. Rather than remembering that operation code `X'43'` copies a byte from memory into the right end of a general register, a *mnemonic operation code* gives a simple indication of what the operation code does. (The term "operation code" is often abbreviated "opcode".) The opcode `X'43'` has mnemonic "IC", which stands for "Insert Character".
3. Symbols let you name areas of memory and other objects in your program.
4. Diagnostic messages warn you about possible errors and oversights.
5. The Assembler converts data from convenient external representations into internal forms.
6. It creates relocatable object code to be combined with other programs by the linker.
7. Using macro-instructions, you can define your own instruction names to supplement existing instructions, and your own macro instructions can make use of previously defined sequences of statements, including other macros!
8. It provides lots of other helpful information such as cross-references of symbols, registers, and macros.

Terms and Definitions

Assembler
A program that converts Assembler Language statements into machine language, in the form of an object module.

assembly time
The time when the Assembler is processing your program's statements, as distinct from the time when the machine language instructions created from your Assembler Language program are executed by the processor.

code
An informal term for groups of Assembler Language statements.

execution time
The time when your program has been put in memory by the Program Loader and given control. This may happen long after assembly time.

Job Control Language
The statements needed to tell your Operating System how to process your program through the assembly, linking, and execution phases. “JCL” for short.

Linker
A program that converts and combines object modules and load modules into an executable “load module” format ready for quick loading into memory by the Program Loader. The term “Linker” can describe several programs:

Binder
The z/OS program that can generate load modules (and a newer form, *program objects*) as well as place the linked program directly into memory.

Linkage Editor
The predecessor to the z/OS Binder; its functions are included in the Binder. A Linkage Editor is used on z/VSE.

Loader
This can have several meanings:

- On z/VM systems, a program that can link object modules directly into memory for execution, or generate a relocatable “MODULE”.
- On older OS/360 systems, a program that links object and load modules into memory for execution; now called the “Batch Loader”.

load module
Our generic name for the output of a Linker; a mixture of machine language instructions and data ready to be loaded directly into memory for execution.

macro instruction
A powerful means to encapsulate groups of statements under a single name, and then generate them (with possible programmer-determined modifications) by using the macro-instruction name as an operation field entry.

mnemonic
A convenient shorthand for the name of an instruction. For example, the “Branch and Save” instruction has mnemonic “BAS”.

object code
The machine language contents of an object module.

object module
The machine language information created by the Assembler, used as input to the Linker.

operand
(1) Something operated on by an instruction. (2) A field in a machine instruction statement.

origin
A starting value assigned by you (or by the Assembler) needed to calculate offsets and displacements in your program. Because most programs are relocated, it's rarely necessary to specify an origin.

Program Loader
The component of the Operating System that brings load modules into memory, makes final relocations, and transfers control to your program.

relocation
A procedure used by the Linker and the Program Loader to ensure that addresses in your loaded program are correct and usable.

statement
The contents of the records read and processed by the Assembler. There are four types: comment statements, machine instruction statements, assembler instruction statements, and macro-instruction statements.

statement field
One of the four fields of an Assembler Language statement (other than a comment statement). They are the name, operation, operand, and the remarks fields. Which fields are required and/or optional depends on the specific statement.

Programming Problems

Problem 6.1.(2)+ Write, assemble, link, and execute a short program (like the one in Figure 33 on page 81) that will print your name. Look through the printed output from the job, and determine which parts were printed by the Assembler, the Linker, and the executed program. (If your name contains apostrophes (like `O'Brien`), you must type a pair of them wherever you want to print one, as in `O''BRIEN`.) Observe what is produced by the Assembler for each type of statement.

Problem 6.2.(2)+ Using your solution to Problem 6.1 as a template, write and execute a program that will generate a noncontroversial, culturally-sensitive, nonpolitical message such as

```
Message = C'Hello, World!'
```

7. Self-Defining Terms and Symbols

```
777777777777
777777777777
77        77
          77
         77
         77
        77
        77
       77
       77
      77
      77
```

We now investigate two important features of the Assembler Language, *self-defining terms* and *symbols.* Each has a numeric value. In a self-defining term, the value is constant and inherent in the term, so you can think of them as different ways to write numbers. Self-defining terms are *not* data! They are just *numbers* that can be written in any of several convenient forms; they all result in 32-bit integer values. Symbols have values assigned by you and by the Assembler.

7.1. Self-Defining Terms

There are four[46] basic types of self-defining term: decimal, hexadecimal, binary, and character. The value of each is treated by the Assembler as a 32-bit two's complement number.

- A *decimal* self-defining term is an unsigned string of decimal digits. 12345, 98, and 007 are examples of decimal self-defining terms. The size of a decimal self-defining term is determined by the fact that 32 bits are allotted by the Assembler to hold its value during assembly. Because it is unsigned, a decimal self-defining term must lie in the range from 0 to $+2^{31}-1$ (2147483647). Thus, +2147483647 and −2147483647 are not valid decimal self-defining terms because they are signed, even though their values can be correctly represented in 32 bits.
- A *hexadecimal* self-defining term is written as the letter “X”, an apostrophe, a string of hexadecimal digits, and a second apostrophe. `X'123456'`, `X'FACED'`, and `X'001B7'` are examples of hexadecimal self-defining terms. The value of a hexadecimal self-defining term must lie in the range from 0 to $+2^{32}-1$, or, between `X'00000000'` and `X'FFFFFFFF'`. If fewer than eight digits are specified, the Assembler assumes that the omitted digits are high-order zeros. If the high-order digit of an eight-digit hexadecimal self-defining term lies between `X'8'` and `X'F'`, the value of the term is negative.

  Because hexadecimal terms represent just a string of bits, their value can be greater than $2^{31}-1$, unlike decimal terms.
- A *binary* self-defining term is written as the letter “B”, an apostrophe, a string of binary digits, and a second apostrophe. `B'110010'`, `B'0001'`, and `B'1111111100001100'` are examples of binary self-defining terms. Because 32 bits are allotted for the value of a self-defining term, at most 31 binary digits may follow the first 1-bit. (For example,

[46] A fifth type of self-defining term, the *Graphic* type, requires invoking the Assembler with the DBCS option. Its use is beyond the scope of this section, but we'll meet it again in Chapter VI, Section 26.4.

B'00000000000000000100000000000000000000000' has 41 digits, but only 24 significant digits follow the first 1.) If fewer than 32 digits are specified, the Assembler assumes that the omitted digits are high-order zeros.

The value of a binary self-defining term must lie in the range from 0 to $2^{32}-1$. The value of a binary self-defining term is negative if the leftmost significant bit of the 32-bit digit string contains a 1-bit.

We will see in Chapter 4 that embedded blanks can be used in decimal, binary, and hexadecimal constants to improve readability. However, embedded blanks cannot be used in self-defining terms of those three types.

- A *character* self-defining term is written as the letter “C”, an apostrophe, a string of up to four characters (except for two special cases to be described momentarily), and a second apostrophe. Thus, C'A', C'...', and C'A•B' are valid character self-defining terms. (Remember that we are using “•” to represent a blank.) This last example, in which a blank appears, is the one exception to the rule mentioned in the previous section that stated that the operand field is terminated by the first blank column after it starts: if the blank is part of a character string enclosed in apostrophes, as in a character self-defining term, it doesn't terminate the field but is part of the operand. A blank terminating the operand field must appear *outside* of a character string enclosed in apostrophes.

  The two special cases concern the apostrophe (') and the ampersand (&). Since apostrophes are used to delimit the character string, we need a way to get an apostrophe *into* the generated character string. (The ampersand has special uses in macro-instructions.) We represent a *single* apostrophe or ampersand in a character string by a *pair* of apostrophes or ampersands. A character self-defining term containing a single apostrophe or a single ampersand is written C'''' or C'&&'. This can lead to cryptic but valid forms like C'''''''' (for the three characters ''', giving a term with value X'007D7D7D'), and and C'&&''&&' (for the three characters &'&, giving a term with value X'00507D50'). A pair of apostrophes is entered as *two* characters, and should not be confused with the quotation mark (″), which is a single character.[47]

Character self-defining terms use the EBCDIC character representation described next.

Exercises

7.1.1.(2)+ Which of the following are valid self-defining terms? If you think a term is invalid, explain why; otherwise, give the hexadecimal value of the term.

```
(1) 00000012345
(2) B'10101010101010101'
(3) X'0000B4DAD'
(4) X'B4DAD0000'
(5) +65535
(6) B'00000000000111100001111000011110 1'
(7) B'10101101000111100001111000011110 1'
```

7.1.2.(1) The maximum value of a decimal self-defining term is $2^{31}-1$, while the maximum value a binary or hexadecimal self-defining term is $2^{32}-1$. Why are they different?

[47] Unfortunately, people sometimes call the apostrophe or single quote a “quotation mark” or “single quotation mark”. Calling a quotation mark a “double quote” or “″” doesn't help either, because it might be understood to mean a pair of apostrophes.

7.2. EBCDIC Character Representation

The value assigned to a binary, decimal, or hexadecimal self-defining term is clear, as they are familiar bit patterns. But what value should we give to a character self-defining term? This depends on the internal representation or *code* defined for characters. We could decide that the value of `C'A'` should be the same as `X'0A'`, or `X'41'`, or `X'74'`, or `X'A1'`, or even `X'C1'`.

In System z the conventional character code is called the "Extended Binary Coded Decimal Interchange Code", or EBCDIC for short.[48] Each character is represented internally by an eight-bit number — two hexadecimal digits — as indicated in Table 13. The internal bit patterns that represent external characters are a matter of choice; any mutually agreeable set is about as good as any other. The Extended BCD code, or EBCDIC, is the code defined by the designers of System/360 for communicating with character-sensitive components of the computer such as the CPU, printers, graphic display devices, etc. We will see other important character encodings in Chapter IV, Section 12.8, and again in Chapter VII, Section 26.

This table shows the EBCDIC representation used by the Assembler, "Code Page 037". (There are many other EBCDIC code pages used around the world.)

Table 13. Assembler Language EBCDIC character representation

Char	Hex	Char	Hex	Char	Hex	Char	Hex
Blank	40	e	85	y	A8	S	E2
.	4B	f	86	z	A9	T	E3
(	4D	g	87	A	C1	U	E4
+	4E	h	88	B	C2	V	E5
&	50	i	89	C	C3	W	E6
$	5B	j	91	D	C4	X	E7
*	5C	k	92	E	C5	Y	E8
)	5D	l	93	F	C6	Z	E9
-	60	m	94	G	C7	0	F0
/	61	n	95	H	C8	1	F1
,	6B	o	96	I	C9	2	F2
_	6D	p	97	J	D1	3	F3
#	7B	q	98	K	D2	4	F4
@	7C	r	99	L	D3	5	F5
'	7D	s	A2	M	D4	6	F6
=	7E	t	A3	N	D5	7	F7
a	81	u	A4	O	D6	8	F8
b	82	v	A5	P	D7	9	F9
c	83	w	A6	Q	D8		
d	84	x	A7	R	D9		

In Table 13 we see that the value associated with the character self-defining term `C'/'` is the same as that of the hexadecimal self-defining term `X'61'`, the binary self-defining term `B'1100001'`, and the decimal self-defining term 97. Similarly, the character self-defining term `C''''` has the same value as the hexadecimal self-defining term `X'7D'`, and `C'&&'` has the same value as `X'50'`. Which type of term you choose is largely a matter of context; in some places, certain types will be more natural than others.

48 Occasionally it is even called BCD. That term is normally used to denote an older six-bit character code or even a 4-bit encoding of decimal digits; the eight-bit *Extended* BCD code is used to represent characters on System z.

The *value* of a character self-defining term is determined by right-adjusting the EBCDIC codes of the characters in a 32-bit field, and filling with zero bits at the left end if needed. Thus, the value of C'A' is X'000000C1', and the value of C'ABC' is X'00C1C2C3'.[49]

The characters shown in Table 13 on page 87 are the portion of the EBCDIC character set used in the Assembler Language (except in character self-defining terms and character constants, where all 256 possible characters are allowed). The codes for other characters are defined in the *z/Architecture Principles of Operation*. It is worth remembering that the EBCDIC code for a blank space is X'40'.

Exercises

7.2.1.(2)+ Which of the following are valid self-defining terms? If you think a term is invalid, explain why; otherwise, give the hexadecimal value of the term.

```
(1) C'#@$'
(2) C'''''
(3) C'•A•B'    (one leading blank)
(4) C'RUD'
(5) C'12'
(6) C'•••12'   (three leading blanks)
```

7.2.2.(2)+ Give (in hexadecimal) the value of each of the following character self-defining terms: (1) C'&&', (2) C'75', (3) C'''', (4) C'C''', (5) C'0', (6) C'SDT'.

7.2.3.(3) Another widely used character representation is the *United States of America Standard Code for Information Interchange*, or ASCII. Determine the ASCII representation of the characters in Table 13 on page 87.

7.2.4.(2)+ Give (in hexadecimal) the value of each of the following self-defining terms: (1) C'''''''', (2) 1000, (3) B'01000', (4) C'&&''&&', (5) C',', (6) C'A=B'.

7.2.5.(3)+ For each of the following values, display all four self-defining terms that may be used to represent it. (1) 64, (2) 245, (3) C'&&', (4) X'405C', (5) X'F9F9F9F9', (6) B'110001011101100111010001'.

7.2.6.(1) What EBCDIC character would be represented by the bit pattern in the byte illustrated in Figure 6 on page 43?

7.2.7.(1) Show the hexadecimal value of each of the following self-defining terms:

```
(1) B'110010110000010111010110'
(2) C'A&&B'
(3) 54721
(4) X'B00B00'
```

7.2.8.(1) Consider the 16 bits 1101000111000101:

1. Write them as four hexadecimal digits.
2. Assuming the bits represent an unsigned (logical) binary number, give its value.
3. Assuming the bits represent a signed binary number in the two's complement representation, give its value.
4. Write them as two EBCDIC characters.

7.2.9.(1) Give the hexadecimal value of these self-defining terms:

[49] In some cases, you might want to use a different character set in character terms. It is possible that the Assembler might assume that your characters are represented in EBCDIC, and generate the wrong value. If you specify the TRANSLATE and COMPAT(TRANSDT) options, Assembler will use your chosen representation for character terms. (See the *High Level Assembler Programmer's Guide* for details.)

1. B'110010111000010111011011001'
2. C'R&&Z'
3. 51401

7.2.10.(2)+ Give the value in hexadecimal of these self-defining terms:

(1) B'01110101100010'
(2) C'''+'
(3) 10010

7.3. Symbols and Attributes

Many programming problems can be greatly simplified by using symbols. If this were not so, we might try to dispense with Assemblers and be content with producing programs consisting of strings of hexadecimal digits; thus we would write the hex digits X'580064EC' instead of a machine instruction statement containing symbols.

Symbols are more interesting than self-defining terms: they let you assign meaningful names to parts of your program. You can give the name **PLUS1** to an area containing the constant +1, and the name **READ** to an instruction that reads data.

Three types of symbols are used in the Assembler Language: ordinary symbols, variable symbols, and sequence symbols. The last two are used only in macro-instructions and in conditional assembly, so we won't say more about them here.

There are two types of ordinary symbols: internal and external. External symbols are used during linking to communicate with other programs (and are part of the object module, as we'll see in Chapter X, Section 38), while internal symbols are used only during the assembly, and do not appear in the object module.[50]

For now, we assume that all symbols are internal symbols. A word of caution: if you have done some programming in a high-level language, you may be inclined to think of symbols as *variables*. They aren't; the differences are described in Section 7.7 on page 94.

A symbol is a string of letters or digits, the first of which must be a letter. The characters “$”, “_”, “#”, and “@” are considered to be *letters* in the Assembler Language.[51] These special characters are *not* allowed in symbols:

```
( ) + - * / = . , ' & blank
```

Early Assemblers restricted symbols to at most eight characters, which is why the “customary” operation field of a statement begins in column 10. HLASM allows mixed-case symbols up to 63 characters long, but there is no difference between upper and lower case letters. Thus, **NAME**, **Name**, and **name** all refer to the same symbol.

The following are all valid symbols.

```
A          Agent086     A1B2D3C4     _The_End
#235       OO@H         ApoPlexy     The_Utter_Final_Bitter_End
James      KQED         Prurient     EtCetera
$746295    Wonka        ZYZYGY99     Close_Files
```

50 Internal symbols are added to the object module if you specify the Assembler's TEST option, but that option is little used now. The ADATA option is preferable, because it generates a SYSADATA “side file” containing much more useful information that can be used by other programs like debuggers.

51 If there's any chance your program might be sent (or read or printed) outside the United States, avoid the “national” characters #, @, and $. They may look different in other countries, or may even have different EBCDIC representations. Other characters usable in Assembler Language symbols — those in Table 13 on page 87 — always have the same EBCDIC representations.

Note that the first character of the symbol OO@H must be the letter "O" and not the digit zero "0". (A good reason to avoid using symbols starting with the letter O!)

The following are not valid symbols, for the reasons given.

```
$7462.95     (decimal point not allowed)
Bond/007     (special character / not allowed)
Set Go       (no blanks allowed)
Ten*Five     (contains the special character *)
C'Wonka'     (no apostrophes allowed)
2Faced       (doesn't begin with a letter)
An_Absurdly_Long_Symbol_With_No_Use_Other_Than_To_Illustrate_Excessive_Symbol_Length (!)
```

Several numeric quantities called *attributes* are associated with a symbol. Symbols have six primary attributes: value, relocation, length, type, scale, and integer.[52] Of these, the value and length attributes are most important; the rest will be described as needed. The length attribute is especially useful, and we'll see how it's defined when we examine constant definitions in Section 11.

- The Assembler assigns numeric values to the attributes of a symbol when it encounters the symbol as the name field entry in a statement. We say that a symbol has been *defined* when numeric values have been given to its value, relocation, and length attributes. These three attributes, like all other numeric attribute values, are *always* nonnegative.
- This terminology is clumsy: rather than the "numeric value of the value attribute" of a symbol, we simply say the "value of the symbol". Similarly, the "numeric value of the relocation attribute" of a symbol is its "relocatability". We say that a symbol whose relocation attribute is nonzero is *relocatable*, and a symbol whose relocation attribute has a zero value is *not relocatable,* or that it is *absolute.*[53]
- We call the "numeric value of the length attribute" of a symbol its "length attribute". It depends on the type of statement named by the symbol. Occasionally someone refers to the "length" of a symbol when its length attribute is meant; but the length of a symbol might be misunderstood to mean the number of characters in the symbol itself, which is rarely interesting. The length *attribute* is different, and is *very* useful.

  For example, while the symbol **A** is one character long, it could have length attribute 133!

Symbols are used mainly as names of places in the program. For example, in Figure 30 on page 78, the symbol **LOAD** is the name of the instruction. Similarly, in the machine instruction statement

```
GETCONST L      0,4(2,7)
```

the symbol **GETCONST** is the name of an area of the program containing a machine instruction. In the Assembler instruction statement

```
TEN      DC     F'10'
```

the symbol **TEN** is the name of a word area of the program where the Assembler will place a binary integer constant with decimal value 10.

In the macro-instruction statement

```
EXIT     RETURN (14,12),T
```

the symbol **EXIT** names the area of the program containing the set of instructions generated by the RETURN macro-instruction.

---

52 Conditional assembly supports additional attributes: Assembler, Count, Number, Defined, Opcode, and Program. The Assembler, Opcode, and Type attributes have nonnumeric values.

53 A useful definition of the relocation attribute is that a symbol that names a place in a program is relocatable; details are given in Section 7.6 on page 93. A convenient image is to think of the relocation attribute of a symbol as its color: the Assembler assigns the same color to all symbols having the same relocation attribute, and no color to absolute symbols.

No symbol can be given a value in a comment statement.

Remember: the attributes of the symbols, and the symbols themselves, exist only at assembly time. They help in producing the object program; *internal symbols and their attributes are discarded when the assembly is complete.*[54]

Exercises

7.3.1.(2)+ Which of the following are valid symbols? If you think a symbol is invalid, explain why.

```
(1) SuperBOY
(2) Captain Major
(3) KillerWhale
(4) Send400$Soon
(5) #@$!
(6) 4Hundred$Sent
(7) ?
(8) (Eight)
(9) @9AM
```

7.3.2.(2) Some Assemblers (for processors other than System z) allow you to define a symbol as a string of alphanumeric characters at least one of which must be a letter (it needn't be the first character). Can you think of any reasons why the designers of the Assembler Language decided not to allow this form of symbol?

7.4. Program Relocatability

Understanding the value and relocation attributes of a symbol is usually not very important. You can write lots of Assembler Language programs without ever having to know how and why the Assembler uses these attributes. When things go wrong (and because things *will* go wrong), it is worth understanding some basic features of value and relocation.

The most important part of the Assembler's task of converting a program from Assembler Language statements to machine language code is determining the relative positions of *all* parts of your program. To do this, the Assembler constructs an accurate model of the program as it will eventually reside in memory when it is executed.

This model is necessarily incomplete, for two reasons:

1. The Assembler normally has no way to know where the program will eventually be placed in memory by the Program Loader.
2. There is no way for the Assembler to know the relationship of the program it is assembling to other programs that will be combined with it in the load module produced by the Linker.

Methods for handling the second reason will be treated when we discuss external linkages and subroutines in Chapter X, Sections 37 and 38.

Because the Assembler cannot determine in advance what memory addresses will eventually hold the program, it must produce a machine language program that will work correctly no matter where it is placed at execution time. That is, the program must be *relocatable.* Thus, in building its model of the final form of the program, the Assembler only needs to determine the *relative* positions of the parts of the program it is assembling.

The Assembler doesn't know where the program will eventually be placed in memory, so it does the next best thing:

---

[54] This information can be saved in a SYSADATA “side file” when you specify the Assembler's ADATA option.

1. It assumes that the program starts at some arbitrary (or programmer-specified) origin, and generates instructions and data based on that assumption.
2. It includes enough information about its assumptions in the object module, so the Linker and the Program Loader can tell (a) what starting location was assumed, and (b) what parts of the program will contain or depend on actual memory addresses at the time the program is executed.
3. By computing the difference between the program's assembly-time starting location assumed by the Assembler, and its true starting address assigned at execution time by the Supervisor, the Program Loader can supply ("relocate") the necessary true addresses used at execution time.

In practice, very few parts of a program depend on knowing actual addresses; these will almost always involve the use of *address constants*; we'll introduce them in Section 12.2 on page 147. Many programs can be written to contain *no* address-dependent information.

7.5. The Location Counter

To help clarify the differences between assembly and execution times, we will make a careful and important distinction between *locations* and *addresses*.

- *Locations* refer to positions in the Assembler's model of the program at assembly time.
- *Addresses* refer to the positions in memory, at execution time, where the various parts of the program reside.

> **Locations and Addresses**
>
> Locations are used at *assembly* time; addresses are used at *execution* time.

The relationship between locations and addresses is close; they differ at most by a single constant value, the difference between the Assembler's assumed assembly-time starting location and the Supervisor's assigned execution-time starting address. This difference is handled by the Program Loader when it relocates the program just before execution, so we don't have to worry about this at assembly time. [55]

To assign locations to the various parts of your program as it is assembled, the Assembler maintains an internal counter called the *Location Counter*, or LC. The initial value of the LC is the "initial location" or "assumed origin" specified on the START statement (see Section 6.4 on page 79); or, if no initial location is specified, the Assembler assigns an initial LC value of zero.

As the Assembler reads your program, it determines how many bytes will be required in the program for the instruction or data generated for each statement. It adds this number to the LC, and then reads and processes the next statement. In this way, the Assembler determines the location and length of each part of the program.

It is important to understand the difference between the Assembler's Location Counter and the CPU's Instruction Address. The LC is a counter used by the Assembler at *assembly* time to determine positions within a program; it goes away when the Assembler is removed from memory at the completion of an assembly. The IA is a part of the CPU's PSW, and contains the address of the next instruction to be fetched at *execution* time; it is *always* in use whenever *any* program is being executed.

[55] The Assembler puts the assumed origin into the object module to help the Linker adjust addresses correctly.

Exercises

7.5.1.(3)+ In the following program segment, determine (1) the value attributes of all symbols, and (2) the LC value at the time each statement is read by the Assembler. The length of the generated instructions and data are given in the comment field of each statement.

```
EX7_5_1  START X'5000'        0 bytes generated
         BASR  6,0            2 bytes generated
BEGIN    L     2,N            4 bytes generated
         A     2,ONE          4 bytes generated
         ST    2,N            4 bytes generated
DUMMY    DS    XL22          22 bytes generated
N        DC    F'8'           4 bytes generated
ONE      DC    F'1'           4 bytes generated
```

We will revisit this program fragment in Section 10.

7.6. Assigning Values to Symbols

Instructions and data are given names by writing symbols as the name field entry of the statement. When the Assembler encounters such a symbol, it enters it into a *Symbol Table* containing the program's symbols and their attributes.

1. The value attribute (or simply, the value) of the symbol is determined from the contents of the LC at the time the statement was processed, *before adding the length of the generated instruction or data*.
2. The relocation attribute will be nonzero, to indicate that the symbol is relocatable. (We will see shortly how to define absolute symbols that are not relocatable.)
3. The length (in bytes) of the generated instruction or data is assigned as the value of the length attribute (in most cases).

There are, of course, occasional minor exceptions to these general rules.

There is a simple test to determine whether an internal symbol is relocatable: add a constant to the initial value of the LC, and re-assemble the program. If the value of the symbol increases by exactly the same amount, then the symbol is relocatable. If the value doesn't change at all, the symbol is absolute.

The names of instructions and data areas in a program are relocatable; these are the most frequent uses of symbols. The numeric value of the relocation attribute of a symbol is assigned by the Assembler, and can be determined from the Assembler's *External Symbol Dictionary*, another part of the object module.

To illustrate how values are assigned to symbols, suppose that when the statement named **GETCONST** (on page 90) is read by the Assembler, the value of the LC is X'0007B6'. Then the symbol **GETCONST** would appear in the Symbol Table with value X'0007B6'; it would be relocatable; and because the statement specifies an RX-type instruction, the length attribute will be 4. Before reading the next statement, the Assembler increments the LC by the length of the instruction, so that its value will then be X'0007BA'.

Similarly, if the sample statement named **TEN** (on page 90) was encountered when the LC value was X'012D88', then the value of the symbol **TEN** would be X'012D88'; it would be marked as relocatable; and its length attribute would be 4. The LC value after incrementing would be X'012D8C'.

To define an absolute symbol, we use the “EQU” assembler instruction statement:

```
symbol   EQU   self-defining term
```

This statement causes the value of the self-defining term to be assigned as the value attribute of the symbol. (More about the EQU assembler instruction is in Section 13.3.) Thus, the statement

```
ABS425   EQU   425
```

defines the symbol **ABS425** by assigning a value of 425 (X'000001A9'), a relocation attribute of zero, and (for want of anything better) a length attribute of one. The symbol **ABS425** is simply the name of a number!

Absolute symbols give you great freedom and flexibility in writing your programs. We will find many ways to use absolute symbols whose values do *not* change if the initial LC value is changed.

Exercises

7.6.1.(1) Why can a symbol not be given a value in a comment statement?

7.6.2.(1) The symbol **TEN** on page 89 will be assigned a length attribute of 4 by the Assembler. What is the length of the symbol?

7.7. Symbols and Variables

In Assembler Language, we make some important distinctions in terminology. In high-level languages such as FORTRAN, COBOL, PL/I, and C, symbols are normally used to name *variables*: you can assign new values to them as the program executes. Thus, you might write

```
BAD = GOOD + 7*(LOG(BETTER)/SQRT(BEST)) ;  /* Assign new value to BAD */
```

and understand it to mean "evaluate the quotient of the results of the LOG and SQRT functions, multiply that by 7, add the result to the current value of the variable GOOD, and assign the result as the new value of the variable BAD." Assembler Language doesn't work this way! The value of a symbol is *not* the value of a variable of the same name.

Assembler symbols

Assembler Language symbols are *not* variables. There are no "variables" in the Assembler Language we're describing![56]

Some of the differences in the meanings of symbols in high-level languages and Assembler Language are shown in Table 14.

Assembler Language	High-Level Languages
Used only at assembly time	Can be thought of as existing at execution time
Names of places in a program	Contain execution-time values
Contents of memory has a "value"	Variable has a "value"
The name has a "location" value used by the Assembler to lay out and organize the program	The name is thought of as naming the value of a variable

Table 14. Differences between Assembler Language and high-level language symbols

We will have more to say about this in Section 13.8 on page 173.

[56] The conditional assembly language *does* have variable symbols, but that topic is beyond what we're discussing now.

Terms and Definitions

defined symbol
A symbol is defined when the Assembler assigns values to its value, relocation, and length attributes.

EBCDIC
Extended Binary Code Decimal Interchange Code. Used to assign numeric values to characters. There are many EBCDIC encodings; they assign different values to some characters, but all the alphabetic, numeric, and other characters used in the Assembler Language listed in Table 13 on page 87 are invariant across EBCDIC encodings, except for the characters "$", "@", and "#".

Location Counter (LC)
A counter used by the Assembler at assembly time to build its model of the relative positions of all components of an assembled program.

relocatable
A property of a program allowing it to execute correctly no matter where it is placed in memory by the Program Loader.

relocation
Actions performed by the Linker and Program Loader to ensure that a program in memory will execute correctly no matter where it is loaded. This may require assigning true execution-time addresses to parts of a program.

self-defining term
One of binary, character, decimal, and hexadecimal. Its value is inherent in the term, and does not depend on the values of other items in the program.

symbol
A name known at assembly time, to which various values are assigned. The values may be absolute or relocatable (or even complexly relocatable, as we'll see in Section 8.3).

symbol attribute
Useful information about the properties of a symbol. Attributes include value, relocation, length, type, scale, and integer. (Only the first three attributes are important for our current needs.)

8. Terms, Operators, Expressions, and Operands

```
 8888888888
888888888888
88        88
88        88
 88      88
  88888888
  88888888
 88      88
88        88
88        88
888888888888
 8888888888
```

In this section we will see how to specify components of the operand field entry of various instruction statements.

The operand field entry of a typical machine instruction statement is a sequence of *operands* separated by commas. For example, a typical instruction statement might look like this:

```
symbol   operation  operand1,operand2,...       optional remarks
```

where the name field symbol is often optional, and the operand field may specify zero to many operands.

The operands are formed from *expressions* that are in turn formed by combining *terms* and *operators*.

8.1. Terms and Operators

The basic elements of an expression are *terms*. They can be any of the following items:

- a self-defining term
- a symbol
- a Location Counter reference
- a literal
- a Symbol Attribute reference
  - Length
  - Integer
  - Scale

  We will discuss Integer and Scale attributes later; while they aren't used frequently, they can be very helpful in certain situations.

Terms

Length, integer, and scale attribute references to a symbol are always absolute terms; a symbol can be either absolute or relocatable; literals and Location Counter references are always relocatable. A self-defining term is always absolute.

We have seen how to write symbols and self-defining terms. Literals are special symbols that provide a convenient way to write constants, and we will discuss them in Section 12.6.

A Location Counter reference is written as a single asterisk; it has the attributes of the Assembler's Location Counter, and a length attribute that depends on the type of statement where it is used. The value of * as a Location Counter reference therefore *changes* during an assembly as the LC value changes.

A symbol length attribute reference is written as a letter L followed by an apostrophe followed by a symbol (or an asterisk, for a Location Counter reference).

```
L'SYMBOL    or    L'*
```

is an absolute term whose value is the length attribute of the term following the apostrophe.

The operators used for combining terms are +, −, *, and /, indicating addition, subtraction, multiplication, and division respectively. A term has no sign; however, + and − may be used as *unary* or *prefix* operators, as in +5. In Assembler Language, the asterisk is therefore used in two ways: to denote a Location Counter Reference and as the multiplication operator. The Assembler can distinguish these two uses.

8.2. Expressions

An expression is an arithmetic combination of terms and operators. In the absence of unary plus or minus signs or parentheses, an expression must begin and end with a term, and there must be an operator between each pair of terms. To illustrate, two expressions are

```
GETCONST+X'4A'    and    X+L'X
```

The following expression uses all four types of self-defining term:

```
X'12'+C'.'-B'0001010001'+7
```

Parentheses may be used, as in ordinary mathematical use (and as in familiar procedural languages) to indicate groupings. In evaluating expressions, an expression in parentheses is treated as a term. Thus

```
(A+2)*(X'4780'-JJ)
```

is an expression that is the product of two subexpressions, each of which has two terms and one operator.

Syntactically, an expression may not contain two multiplication or division operators in succession, or an addition or subtraction operator followed by a multiplication or division operator. For example:

```
*+2                    valid because * is a Location Counter reference
-A, +A                 are valid uses of unary + and -
A++B, A--B, A+-B, A-+B   are valid (second + and second - are unary operators)
A/+B, A/-B, A*+B, A*-B   are valid (+ and - are unary operators)
A+/B, A-/B, A+*B, A-*B   are invalid
A**B, A*/B, A/*B, A//B   are invalid
```

Some syntactically valid expressions might not be evaluatable if either or both terms is relocatable (to be described shortly).

An easy way to determine the validity of expressions with sucessive operators is to parenthesize each operator with its immediately following term, so that A++B becomes A+(+B); because the unary + in (+B) is equivalent to B, A++B is evaluated as A+B. (See Exercise 8.2.3.)

Exercises

8.2.1.(2) What would you expect to be the result of A--B, A+-B, and A-+B?

8.2.2.(1) What is the value of the expression X'12'+C'.'-B'0001010001'+7?

8.2.3.(2)+ Determine the syntactic validity of each of the following expressions; and if the expression is valid, show its simplified form.

a. A+-+-B
b. A*--B
c. A-*-B
d. A---B
e. --A-++B

8.3. Evaluating Assembly-Time Expressions (*)

The rules for evaluating expressions are familiar, with one or two minor exceptions, so it's no surprise that the Assembler evaluates 2+3 as 5.

Remember: we are describing the Assembler's evaluation of *assembly-time* expressions involving the values of assembly-time symbols and other terms. This is entirely different from most high-level languages, where an expression like A+B in a statement is evaluated at *execution* time, using the values of the execution-time variables A and B.

The details of the rules can be rather complicated, so don't try to grasp everything on a first reading. The examples on page 100 will help to illustrate the rules.

1. Each *term* (along with any preceding unary operator) is evaluated to word precision, 32 bits. The relocation attribute of each term is noted, so that the relocation attribute of the entire expression can be evaluated also, as described in rule 10 below.
2. Inner parenthesized subexpressions are evaluated first, using 32-bit two's complement arithmetic. The resulting value is used in computing the rest of the expression. Thus in

   ```
   (X'100'+2*(ABS425-420))+1
   ```

   where ABS425 has value 425 (as defined on page 93), the subexpression (ABS425-420) would be evaluated first. The value of the whole expression is X'0000010B', and is absolute.
3. Multiplications and divisions are done *before* additions and subtractions. Thus the value of the expression just given would be evaluated as (X'100'+(2*(5)))+1 and not as ((X'100'+2)*(5))+1. Multiplication and division operators may not be combined, as in /* and */.
4. Relocatable terms or subexpressions may *not* occur in multiplication or division operations.
5. Operations are performed in left-to-right order within a group of operations of the same priority. Thus 5*2/4 means the same as (5*2)/4, not 5*(2/4); similarly, 5/2*4 means the same as (5/2)*4, not 5/(2*4).
6. Multiplications yield a 64-bit result, of which the rightmost 32 bits are kept, and the high-order (leftmost) 32 bits are discarded. Significant bits can be lost if the product is too large.
7. Division always yields an integer result; the Assembler always discards remainders when evaluating expressions. Thus 5*2/4 has value 2, and 5*(2/4) has value zero. *Division by zero is permitted*, and the result is simply set to zero.

8. Negative quantities are carried in two's complement representation.
9. When the expression has been completely evaluated, the result is in 32-bit two's complement form.
10. The relocation attribute of the result is found as follows; assume that the symbol A is relocatable:
    - If there is an even number of relocatable terms appearing in the expression and they are *paired* (that is, they have the same relocation attribute appearing with opposite signs) so that a change in the relative origin assigned to the program has no effect on the value of the expression, then the expression is *absolute.* For example, A-A+2 is an absolute expression with value 2.
    - If there is one remaining unpaired term not directly preceded by a minus sign, then the expression is *simply relocatable,* and it has the relocation attribute of the unpaired term. For example, A+2 is a simply relocatable expression.
    - If there is more than one remaining unpaired relocatable term, or if the remaining term is preceded by a minus sign, the expression is *complexly relocatable.* Intentional use of complexly relocatable symbols is extremely rare. For example, 2-A is a complexly relocatable expression. (Some later examples will show how complex relocatability can happen, so don't worry if this seems obscure.)

In general, you can determine the relocatability of an expression roughly as follows: first, compute the value of the expression. Second, add some constant to the initial value of the LC, which will cause the values of relocatable symbols to change. Third, recompute the value of the expression using the new values of the symbols. If the new value of the expression is identical to the old value, the expression is absolute; if the values differ by the amount added to the LC, the expression is simply relocatable; otherwise it is complexly relocatable.

To summarize the rules for combining terms, let A and R represent respectively an absolute and a simply relocatable expression. The rules for combining terms are summarized in Table 15.

An expression of this form	is
A+A, A-A, A*A, A/A	absolute
R+A, R-A, A+R	simply relocatable
R+R, A-R	complexly relocatable
R*A, A*R, R/A, A/R, R*R, R/R	forbidden
R-R	absolute or complexly relocatable

Table 15. Expressions with absolute and relocatable terms

R−R is absolute only if both expressions have the same relocation attribute. Because this will almost always be true, we assume (until further notice) that expressions of the form R−R are absolute. We'll give a precise definition of the relocation attribute in Chapter X when we discuss external symbols.

Machine instruction statement operands may never be complexly relocatable.

Exercises

8.3.1.(2)+ Suppose R stands for an arbitrary relocatable expression, and A stands for an arbitrary absolute expression. State which of the following expressions are and are not valid in machine instruction statement operands.

```
(1) R+R   (2) A+R   (3) R+A   (4) A+A   (5) R-R   (6) A-R   (7) R-A   (8) A-A
(9) R*R  (10) A*R  (11) R*A  (12) A*A  (13) R/R  (14) A/R  (15) R/A  (16) A/A
```

8.3.2.(2) Rule 7 on page 98 states that the Assembler always discards remainders in evaluating expressions. Does this mean that a program cannot compute a remainder? Explain.

8.3.3.(2)+ The last row of Table 15 says that R-R can be complexly relocatable. How can the difference of two simply relocatable symbols be complexly relocatable?

8.4. Examples

For these examples, we assume that

- ABS425 is an absolute symbol of value 425 (or X'000001A9'),
- the value of the Location Counter is X'00011D46',
- REL1 is a relocatable symbol of value X'00010A20',
- REL2 is a relocatable symbol of value X'00012345' having length attribute 6, and
- The Location Counter, REL1, and REL2 have the same relocation attribute.

1. 5*2/4 = 10/4, an absolute expression of value X'00000002'.
2. 16*16*16*16*16*16 is an absolute expression of value X'01000000'.
3. 6-ABS425 has value X'FFFFFE5D', and is absolute.
4. (REL2-REL1)/(ABS425-B'011111') is an absolute expression of value X'00000010'.
5. REL2+C'-'+*+L'REL2-* is a relocatable expression of value X'000123AB'.
6. 2*REL2-REL1 is an invalid expression, because a relocatable term (REL2) occurs in a multiply operation. (If the Assembler *was* able to evaluate the expression, it would be simply relocatable, and have value X'00013C6A'.)
7. Even REL1*1 and REL1*0 (as well as REL1/1 and REL1/0) are invalid expressions, even though their values are perfectly well defined.
8. (1+(1+(1+(1+(1+(1+(1+(1+2)*2)*2)*2)*2)*2)*2)*2)+1 is an absolute expression, and has value X'200'.
9. *+6 is a relocatable expression of value X'00011D4C'.
10. (REL2-*)*L'REL2 is an absolute expression of value X'000023FA'. Note the two distinct uses of the asterisk!

The example of a machine instruction statement in Figure 30 on page 78 could have been written

```
LOAD    LR    C'45'-(7*X'2A36')+ABS425*B'11111'-235,18/(Q-Q)+3
```

though the gain in clarity is not obvious. More reasonable usage is illustrated in the following statements.

```
*         EXAMPLE 8_4_1
R7        EQU   7
R3        EQU   3
LOAD      LR    R7,R3
```

There is a difference between

1. the *notational* convenience of the symbol R7 defined in the first EQU statement above and intended to mean general register 7,
2. the definition of an absolute *symbol* R7 to have the value 7, and
3. the *use* of the symbol as an operand in the operand field entry of a machine instruction statement where the use of register 7 is intended.

Example 8_4_1 is equivalent to the two below. (The second is considered poor style, for obvious reasons.)

```
*      EXAMPLE 8_4_2              *      EXAMPLE 8_4_3
ZORCH  EQU   3                    R7     EQU   3
ZILCH  EQU   7                    R3     EQU   7
LOAD   LR    ZILCH,ZORCH          LOAD   LR    R3,R7
```

Expressions can also be used to good advantage in EQU statements. For example, suppose we need to define a symbol **NWords** whose value gives the number of words in a table, and we also need a symbol **NBits** whose value is the number of bits in the same table. We could define the symbols in the following way.

```
*      Example 8_4_4         EQU with expressions
NWords EQU   75              Table has 75 word entries
BitsWd EQU   32              Number of bits per word
NBits  EQU   NWords*BitsWd   Number of bits in the table
```

Exercises

8.4.1.(2) What are the values of the symbols **NWords**, **BitsWd**, and **NBits** in Example 8_4_4 above?

8.4.2.(3)+ The following short program segment contains instructions (whose purpose is of no interest for this exercise) whose operand fields contain various expressions. For each expression, determine (1) whether the expression is absolute or relocatable, and (2) the value of the expression. The column headed "LOC" gives the hexadecimal value of the Location Counter for each instruction.

```
LOC        Statement
466        A     L     4,B+X'1C'
46A              BALR  R6,0
46C        B     ST    4,C-A+X-2*(R6/2)
470        C     SLL   5,2*C'-'-C'A'+2
                 USING B-2,R6-2
           R6    EQU   9
474        X     DS    F           Define Symbol X
```

8.4.3.(3)+ Assume that the Location Counter, and symbols **REL1**, **REL2**, and **ABS425** have the value and relocation attributes defined in the examples on page 100. Determine the value and relocation attributes of the following expressions.

```
(1) REL1+C'2'/2
(2) REL1-REL2+ABS425
(3) C'45'-(7*X'2A36')+ABS425*B'11111'-235
(4) (8/(REL2-REL1)/X'107C')+3
(5) ABS425/((REL2-REL1)/X'C701'+3)
(6) *+ABS425*(*-REL1-4900)
```

8.4.4.(2) Assuming that the symbols **REL1**, **REL2**, and **ABS425** have the attributes defined on page 100, determine the *validity* of each of the following expressions. Explain why you think any expression is invalid.

```
(1) -2+ABS425
(2) ((REL1))*2-2*((REL1))
(3) REL1+C'7592'*B'10110'+ABS425
(4) B'10221'+REL2
(5) ABS425*74239661-2
(6) +X'1875'
(7) -*+REL2
(8) **1
```

8.4.5.(3) Assume that the symbols **A** and **B** are simply relocatable with the same relocation attribute, and that they have values X'00172B9E' and X'00173AA6' respectively. Determine the value and relocation attributes of the following expressions.

```
(1) B-A
(2) A+C'.'
(3) (A+X'00FFF')-(B-B'1101011100001')
(4) (B-A)/10
(5) B+C'B'/(B+B'101'-B)
```

8.4.6.(3)+ The symbols **SAM** and **JOE** are simply relocatable with the same relocation attribute, and have values X'00174D0A' and X'0016FB63' respectively. The symbol **BOB** is absolute and has

value X'000003E8'. First, determine the validity of each of the following expressions. Then determine the value and relocation of each of the valid expressions.

```
(1) 2*BOB+2*SAM-2*JOE
(2) BOB+(SAM+BOB)-(JOE+BOB)
(3) 2*(SAM-JOE)/5
(4) SAM-(B'10000'*(X'0010'*(BOB-C'H')))
(5) (2*SAM-2*JOE)/5
(6) 2*(JOE-SAM)/(SAM-JOE)
```

8.4.7.(4) Can you think of any reasons why the designers of the Assembler Language did not allow relocatable terms to appear in multiplications or divisions? Assuming that the final value of the term must be either relocatable or absolute, what modifications would be needed to allow such expressions, as in example 6 on page 100?

8.4.8.(1) The symbols A and B are relocatable, and have values X'00172B9E' and X'00173AA6' respectively. Determine the value and relocation of these expressions:

1. B-A
2. A+C'.'

8.5. Machine Instruction Statement Operand Formats

The operand field entry of a machine instruction statement consists of a sequence of operands separated by commas, and terminated by a blank not enclosed in apostrophes. For example, the operand field entry of the LR machine instruction statement in Examples 8_4_1 through 8_4_3 contains two operands, expressions of value 7 and 3 respectively.

An operand of a *machine* instruction statement has only one of three possible formats:

$expr$ $expr_1(expr_2)$ $expr_1(expr_2,expr_3)$

where “expr” is an abbreviation for “expression”, and the subscripts indicate only that each expr can be different from the others. To repeat: *operands of machine instruction statements have one of these three formats.*

The third operand format has two interesting features. First, the comma between the second and third expressions does *not* terminate the operand; it merely separates the expressions within the parentheses. Second, the first of the expressions within the parentheses, $expr_2$, may sometimes be omitted, so that

$expr_1(,expr_3)$

is a valid form of the third operand format. The Assembler will assume that the omitted expression is absolute and has value zero. The format $expr_1(expr_2,)$ is never valid.

Examples of the first expr format are

```
ABLE     2*(SAM-JOE)/5     X'6D'    TWO+2     *
```

Examples of the second $expr_1(expr_2)$ format are

```
ABLE(4)     X'6D'(POINTER)     P(*-*)     (A-ST)(2+ST)
```

Multiplication is *not* implied in the last example!

Finally, examples of the third $expr_1(expr_2,expr_3)$ format are

```
0(255,12)    8(,3)    X(Y-8,Z/2)    (A-B)(A-B,(A-B))
```

Again, no multiplication is implied in any example.

Depending on the machine instruction, one or more operands may be required; for each operand, one or more of the operand formats may be valid. Also, depending on the type of the instruction,

there may be restrictions on the value and relocation attributes of the expressions in an operand. One of the most important restrictions is that all operands of a machine instruction statement must either be absolute or simply relocatable; no complexly relocatable expressions are allowed.

For example, a typical RR-type instruction (as in the examples on page 100) has two operands: each must be of the form

```
expr
```

For such RR-type instructions, the Assembler requires that the expressions must be absolute and have value between 0 and 15.

Exercises

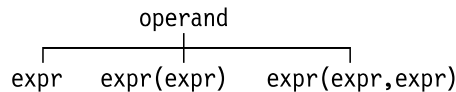
8.5.1.(2)+ For each of the following operands, determine whether it is of the first, second, or third type. If the operand is invalid, explain why.

```
(1) A+B(5)
(2) A+(B+(5))
(3) A+C'('(C')')
(4) A(C',C')
(5) 7+(X'BAD'/B'01101')
(6) (C'(')(C'),',(C'(,)'))
(7) 0-0(0,0)
(8) 0/0(,0*0)
(9) C'''(A)'*C'A('(C')'')'-X'C'*C'X)')
```

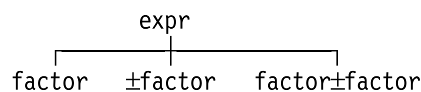
8.6. Details of Expression Evaluation (*)

While the rules for writing specific machine instruction statement operands will be covered in the later sections as new instruction types are introduced, this view of the rules for valid expressions (stated in the previous section) can be summarized in these diagrams.[57]

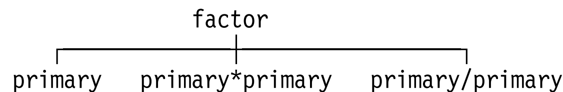
1. An operand can take one of three forms:



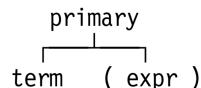
2. An expression can take any of these three items involving a “factor” (this shows how unary + and − signs are described):



3. A factor can take any of these three forms (this shows how multiplication and division have higher priority than addition and subtraction):



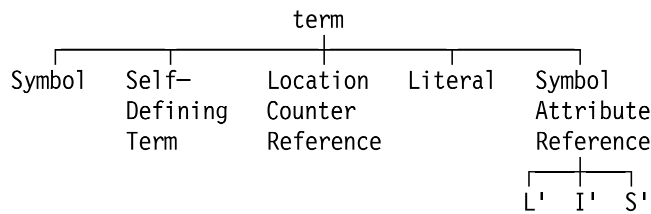
4. A primary is either a term or a parenthesized expression:



5. Finally, a term in an expression is one of the following:

---

[57] These five diagrams are pictorial representations of a notation known as “BNF”, which stands for either “Backus Normal Form” (after John Backus, the leader of the team that created the first FORTRAN compiler in 1957), or “Backus-Naur Form” (after John Backus and Peter Naur, who worked on defining the ALGOL language in 1958-1960.)



We haven't yet described Literals and Symbol Attribute References; they will appear shortly.

The quantities “factor” and “primary” do not appear anywhere in the Assembler Language. They are used here only to help clarify the precedence of multiplication, division, addition, subtraction, and parentheses.

Terms and Definitions

absolute symbol
: A symbol whose value behaves in expressions like a self-defining term. Its value does not change if the assumed origin of the program changes.

complex relocatability
: A property of a symbol or expression whose relocation attribute is neither absolute or simply relocatable.

expression
: A combination of terms and operators to be evaluated by the Assembler.

expression evaluation
: The procedure used by the Assembler to determine the value of an expression.

Length Attribute Reference
: A term whose value is the length attribute of a symbol.

operator
: One of * (meaning multiplication), / (meaning division), + (meaning addition), or − (meaning subtraction). (The Assembler does not support **, which is sometimes used to mean exponentiation.)

simple relocatability
: A property of a symbol or expression whose value changes by the same amount as a change to the program's assumed origin.

Symbol Attribute Reference
: A term whose value is that of a symbol's attribute. The three most important types of Symbol Attribute Reference are length, scale, and integer.

term
: A symbol, self-defining term, Location Counter reference, literal, or symbol attribute reference.

Programming Problems

Problem 8.1.(2) Write and execute some test cases with your Assembler to determine whether it allows you to specify a Length Attribute reference of *any* term, not just for symbols and Location Counter References. Are there any cases that don't work? (Some test cases you might try are `L'2`, `L'(*-10)`, `L'*`, `L'ABS425`, `L'425`, `L'=F'1'`, and `L'L'*`.)

Problem 8.2.(2) What is the length attribute of an expression? Suppose `A` and `B` are absolute symbols with value 5 and 3 respectively, and they both have length attribute 1. Determine the value of each of the following expressions: (1) `L'A*B`, (2) `A*L'B`, (3) `L'(A*B)`. Evaluate them on your Assembler. This code fragment may help you start:

```
A       Equ    5
B       Equ    3
C1      Equ    L'A*B
C2      Equ    A*L'B
C3      Equ    L'(A*B)
```

Try some similar expressions and see what happens.

9. Instructions, Mnemonics, and Operands

```
 9999999999
999999999999
99        99
99        99
99        99
999999999999
999999999999
          99
          99
99        99
999999999999
 9999999999
```

We will now see how to write some machine instruction statements, with various instruction formats and examples of actual code sequences. The instructions in Table 16 and their behavior will be discussed in detail later, so don't worry now about learning the mnemonics, operation codes, or descriptions.

Mnemonics are short abbreviations for a word or phrase describing the action of each operation code. A mnemonic may be as simple as “A” meaning “Add”, or “BXLE”, meaning “Branch on Index Low or Equal”. We will look at several classes of instructions, showing how their operands are written. Abbreviations and notations used to describe operands such as “R_1”, “S_2”, “I_2”, etc., will be explained as we go along.

9.1. Basic RR-Type Instructions

Table 16 illustrates some common RR-type instructions, where “Op” and “Mnem” are abbreviations for “Operation Code” or “Opcode”, and “Mnemonic”.

Op	Mnem	Instruction	Op	Mnem	Instruction
04	SPM	Set Program Mask	05	BALR	Branch And Link
06	BCTR	Branch On Count	07	BCR	Branch On Condition
0D	BASR	Branch And Save	0E	MVCL	Move Long
0F	CLCL	Compare Logical Long	10	LPR	Load Positive
11	LNR	Load Negative	12	LTR	Load And Test
13	LCR	Load Complement	14	NR	AND
15	CLR	Compare Logical	16	OR	OR
17	XR	Exclusive OR	18	LR	Load
19	CR	Compare	1A	AR	Add
1B	SR	Subtract	1C	MR	Multiply
1D	DR	Divide	1E	ALR	Add Logical
1F	SLR	Subtract Logical			

Table 16. Typical RR-type instructions

1. Not all of the 64 available bit combinations between X'00' and X'3F' are used as actual operation codes. For example, IBM has promised not to use X'00' as an operation code.[58]
2. There are many other RR-type instructions, and several other RR-type instruction formats. The examples that follow generally apply to all such instructions.

9.2. Writing RR-Type Instructions

For most RR instructions, the operand field entry in a machine instruction statement is written

R_1,R_2

where the operands “R_1” and “R_2” designate registers.[59] (Some instructions require one or both of the operands to be even numbers, designating even-numbered registers.)

The numeric subscripts “1” and “2” in the quantities “R_1” and “R_2” distinguish the operand being referenced. Using the terms “first operand”, “second operand”, etc. consistently will help you remember what actions are being performed by each instruction.

To explain the notation “R_1,R_2”, refer to the example of a machine instruction statement in Figure 30 on page 78, where the operation and operand field entries were “LR” and “7,3”, respectively. In this case, the “R_1” operand is “7” and the “R_2” operand is “3”. The quantities R_1 and R_2 must be absolute expressions between 0 and 15. Thus, we could just as well have written

```
LOAD     LR    X'7',B'11'
```

For these basic RR-type instructions, the values of the operand field expressions are placed by the Assembler into two adjacent hexadecimal digits, called the “operand register specification digits” in the second byte of the instruction. This second byte was denoted “register specification” in Table 6 on page 52. Table 17 shows the positions of the register specification digits.

opcode	R_1	R_2

Table 17. RR-type instruction

For most RR instructions the R_1 operand specifies the register that at execution time contains the “first operand”. Our notation “R_1” means a number specifying the R_1 digit of an instruction; no reference to general register register 1 (possibly denoted by GR1) is implied. You can of course specify “1” as the value of the R_1 operand!

We can now see the difference between (1) the “operands” of an instruction statement at assembly time, and (2) the “operands” of a machine instruction at execution time. The operands (first meaning) in the operand field entry of the instruction “LR 7,3” are the single characters 7 and 3, whereas at execution time the operands (second meaning) of the LR instruction will be the *data* found in general registers 7 and 3. Table 16 on page 106 shows that the operation code corresponding to the mnemonic LR is X'18', so the two-byte instruction generated by the Assembler would be X'1873'.

Programming with RR instructions is easy. Suppose we wish to compute the sum of the contents of general registers 2 and 14, subtract the contents of GR9 from the sum, and leave the result in GR0. These statements will do the job.

[58] X'00' has not been assigned as a valid opcode for two reasons. First, unused areas of memory are often set to zero when programs are initialized; programs that try to execute “instructions” from those areas will stop immediately with a program interruption for an invalid instruction. (Sometimes, a programmer will purposely insert a X'0000' halfword in a program to force it to stop at an exact position so the contents of registers and memory can be verified.) Also, programs like debuggers sometimes use X'00' as “breakpoints” to halt instruction tracing at a specified place.

[59] Some instructions have only one or even *no* explicit operands!

```
LR    0,2              Copy contents of GR2 to GR0
AR    0,14             Add contents of GR14 to GR0
SR    0,9              Subtract contents of GR9 from GR0
```

The instructions, their actions, and other properties will be described in subsequent sections.

Exercises

9.2.1.(2)+ Which of the following are valid register operands for an RR-type instruction? (1) 0, (2) B'1101', (3) X'11', (4) 4*(X'F2'–C'0')/5+X'E', (5) 4*(X'F2'–C'0')/3+X'E'.

9.2.2.(2) Which of the values in Exercise 9.2.1 are valid operands if the instruction operand requires an even-numbered register?

9.3. Basic RX-Type Instructions

Table 18 shows examples of some frequently-used RX-type instructions. As in Table 16, not all of the 64 available digit combinations between X'40' and X'7F' are used as actual operation codes. Again, you needn't try to remember them here.

Op	Mnem	Instruction	Op	Mnem	Instruction
40	STH	Store Halfword	41	LA	Load Address
42	STC	Store Character	43	IC	Insert Character
44	EX	Execute	45	BAL	Branch And Link
46	BCT	Branch On Count	47	BC	Branch On Condition
48	LH	Load Halfword	49	CH	Compare Halfword
4A	AH	Add Halfword	4B	SH	Subtract Halfword
4C	MH	Multiply Halfword	4D	BAS	Branch And Save
4E	CVD	Convert To Decimal	4F	CVB	Convert To Binary
50	ST	Store	54	N	AND
55	CL	Compare Logical	56	O	OR
57	X	Exclusive OR	58	L	Load
59	C	Compare	5A	A	Add
5B	S	Subtract	5C	M	Multiply
5D	D	Divide	5E	AL	Add Logical
5F	SL	Subtract Logical			

Table 18. Typical RX-type instructions

9.4. Writing RX-Type Instructions

In this and the following section we will introduce some basic concepts, using RX-type instructions as examples.

The format of an RX-type instruction was shown in Table 7 on page 52. We now look at the parts of the instruction in Table 19 and describe Assembler Language techniques for specifying them.

opcode	R_1	X_2	B_2	D_2

Table 19. RX-type instruction

As noted when we reviewed addressing in Section 5.3 on page 63, three components of an RX instruction are used in computing an Effective Address: the index register specification digit X_2, the base register specification digit B_2, and the displacement D_2. The operand field entries may be written in several ways, but they must yield values for the four needed quantities: R_1, X_2, B_2, and D_2. Usually, values for all of these items need not be explicitly given; the Assembler can make assumptions about what to provide in cases where values are not explicitly given. When the Assembler provides values for something, we say that the values were “specified by default” or “specified implicitly”.

The operand field entry of RX-type instructions has the general form

```
R1,address-specification
```

where “address-specification” will be described next. The operand register specification digit R_1 is formed according to the same rules given above for the R_1 and R_2 digits for RR instructions, and must be an absolute expression with value between 0 and 15.

9.5. Explicit and Implied Addresses

For an explicit address, *you* supply the base and displacement; for an implied address, the *Assembler* determines the base and displacement. (Section 10 will show you how it's done.)

Explicit and Implied Addresses

- Explicit: *you* specify the base and displacement.
- Implied: the *Assembler* calculates the base and displacement for you. How this is done is explained in Section 10.

Suppose we wish to specify *explicitly* the values assigned to X_2, B_2, and D_2: then, we write the second operand (the “address-specification”) as

```
D2(X2,B2)
```

which is the third of the possible operand formats described in Section 8.5 on page 102. The instructions in examples 4, 5, and 6 of Section 5.4 on page 65 could be written as shown in Figure 34, where the assembled form is on the left, and the Assembler Language machine instruction statement is in the center; the displacements have the same value in each instruction.

```
430A7468         IC   0,1128(10,7)      D2=1128, X2=10, B2=7
43007468         IC   0,1128(0,7)       D2=1128, X2=0,  B2=7
43070468         IC   0,1128(7,0)       D2=1128, X2=7,  B2=0
```

Figure 34. RX Instruction with explicit operands

Compare the machine language form of these three instructions to the fields in Table 19 on page 108.

The four possible forms of the second operand of an RX instruction are shown below, where we use “S_2” to mean an implied address (which need not necessarily refer to a symbol, as we'll see!).

	Explicit Address	**Implied Address**
Not Indexed	$D_2(,B_2)$	S_2
Indexed	$D_2(X_2,B_2)$	$S_2(X_2)$

Table 20. Operands of RX-type instructions

In the two cases where an *explicit address* is written, each of the quantities D_2, X_2, and B_2 must be an absolute expression; X_2 and B_2 must have value less than 16, and D_2 must have value less

than or equal to 4095=X'FFF'.[60] The not-indexed form of an explicit address implies X_2=0, as we saw earlier; both indexed addresses specify an index digit.

In the two cases where an *implied address* is written, the quantity S_2 may be *either* an absolute *or* a relocatable expression. This means that we can write instructions such as

```
         L     0,ANSWER            Operand forms are R1,S2
         L     0,16                Operand forms are R1,S2
         LA    2,25*40             Operand forms are R1,S2
```

and let the Assembler assign the proper base and displacement; this is the subject of Section 10. Note that the second operand of the first statement is a symbol (that we assume is relocatable), while the second operand of the other two statements is an absolute expression.

For the moment, suppose the Assembler has sufficient information so that the instruction

```
         IC    0,BYTE              Operand forms are R1,S2
```

is translated into the hexadecimal digits 43007468, as in Figure 34 on page 109. Then if the index register is GR10, the instruction

```
         IC    0,BYTE(10)          Operand forms are R1,S2(X2)
```

is translated into the hexadecimal digits 430A7468. In the last example in Figure 34 on page 109 we could have written the second operand with an indexed implied address of the form $S_2(X_2)$, as 1128(7), where the S_2 expression is absolute!

For example, it is common practice to load a small constant into a register using the LA (Load Address) instruction:

```
         LA   2,10                Put 10 in R2
```

and the operand 10 is an absolute implied address. This will almost never lead to difficulties; but to be absolutely safe, you could write instead

```
         LA   2,10(0,0)           Put 10 in R2
```

and the operand now specifies an explicit address.

The *only* way the Assembler can decide among the four forms of address specification in Table 20 on page 109 is (1) by noting whether a left parenthesis follows the first expression (if not, the address is implied), and (2) if there *is* a left parenthesis, by noting whether a comma appears before the matching right parenthesis (if so, the address is explicit). There is of course no effect of commas and parentheses in character self-defining terms.

It helps to remember that implied addresses *almost* always involve relocatable expressions, and explicit addresses *always* involve absolute expressions. Sometimes we accidentally use a relocatable expression where it should have been absolute, or an absolute expression where it should have been relocatable. The Assembler usually (but not always) diagnoses such errors.

The most common form of address specification is an implied address, where the Assembler computes the proper displacement for us. While we have now seen implied addresses in the context of RX-type instructions, they are used in many other instruction types.

Exercises

9.5.1.(2) In Table 20 on page 109, use the rules of Section 8.5 to identify the format of each of the four operands.

9.5.2.(2)+ The following are examples of the second operand of an RX-type instruction (the address-specification). For each operand, determine (1) whether the address is implied or explicit, and (2) whether indexing is specified. Assume that the symbols **A**, **B**, **C** are relocatable with the same relocation attribute, and that the symbol **N** is absolute.

---

[60] In Section 20 we will introduce instructions with signed 20-bit displacements.

```
1. B+X'1C'
2. C-A+B-2(N/2)
3. 2*C'-'-C'A'+2(N+N)
4. B-A((B-A)/2,((B-A)*2))
5. C'A'+A(C','-99)
6. N+N(,N)
```

9.5.3.(2) Assume that each of the operands in Exercise 8.5.1 on page 103 is used in an RX-type instruction. Using the rules in Section 9.5, determine whether the addresses are explicit or implied.

9.6. Typical RS- and SI-Type Instructions

The examples of basic RS-type and SI-type instructions in Table 21 are quite varied in the way you specify their operand fields.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
90	STM	RS	Store Multiple	91	TM	SI	Test Under Mask
92	MVI	SI	Move Immediate	94	NI	SI	AND Immediate
86	BXH	RS	Branch On Index High	95	CLI	SI	Compare Logical Immediate
87	BXLE	RS	Branch On Index Low or Equal	96	OI	SI	OR Immediate
97	XI	SI	Exclusive OR Immediate	88	SRL	RS	Shift Right Single Logical
98	LM	RS	Load Multiple	89	SLL	RS	Shift Left Single Logical
8A	SRA	RS	Shift Right Single	8B	SLA	RS	Shift Left Single
8C	SRDL	RS	Shift Right Double Logical	8D	SLDL	RS	Shift Left Double Logical
8E	SRDA	RS	Shift Right Double	8F	SLDA	RS	Shift Left Double
BD	CLM	RS	Compare Logical Characters Under Mask	BE	STCM	RS	Store Characters Under Mask
BF	ICM	RS	Insert Characters Under Mask				

Table 21. Typical RS- and SI-type instructions

Some instructions (like "Shift Double") require a register operand to be an even number.

9.7. Writing RS- and SI-Type Instructions

We will show the operand field formats for RS-type and SI-type instructions separately, as they are quite different.

The RS-type instruction format is similar to RX-type format, except that the X_2 field is replaced by an R_3 field, so no indexing is performed when Effective Addresses are formed.

opcode	R_1	R_3	B_2	D_2

Table 22. Typical RS-type instruction

The operand fields of Assembler Language instructions specifying RS-type instructions are shown in Table 23 on page 112. There are two forms, one with a single "R_n" operand and the other with two, indicated by RS-1 and RS-2 meaning one or two register operands respectively.

	Explicit Address	Implied Address
RS-1	$R_1,D_2(B_2)$	R_1,S_2
RS-2	$R_1,R_3,D_2(B_2)$	R_1,R_3,S_2

Table 23. Operands of RS-type instructions

Examples of RS-type instructions with explicit and implied addresses are:

```
SRA   11,2                Explicit address (RS-1 form)
SLDL  6,N                 Implied  address (RS-1 form)
LM    14,12,12(13)        Explicit address (RS-2 form)
STM   14,12,SaveArea+12   Implied  address (RS-2 form)
BXLE  4,1,Loop_3          Implied  address (RS-2 form)
```

SI-type instructions are different. The I_2 operand is contained in the second byte of the instruction, as in Table 24:

opcode	I_2	B_1	D_1

Table 24. Typical SI-type instruction

Table 25 gives the operand fields of Assembler Language statements involving SI-type instructions:

	Explicit Address	Implied Address
SI	$D_1(B_1),I_2$	S_1,I_2

Table 25. Operands of SI-type instructions

Examples of SI-type instructions with explicit and implied addresses are:

```
MVI   0(6),C'*'           Explicit S1 address
CLI   Buffer,C'0'         Implied S1 address
```

Exercises

9.7.1.(2) The following are operand fields that could be used in RS- and SI-type instructions. Identify the type of instruction (RS-1, RS-2, or SI) for which they are valid, and the components of the instruction to which each expression applies. State which expressions specify explicit addresses and which specify implied addresses.

(1) `1(2),3`
(2) `4,5(6)`
(3) `7,8,9`
(4) `10,11`
(5) `14,15(16)`
(6) `100,101`

9.8. Typical SS-Type Instructions

Table 26 shows some examples of popular SS-type instructions. The column headed “Len” shows the number of length fields in the instruction.

Op	Mnem	Len	Instruction	Op	Mnem	Len	Instruction
D1	MVN	1	Move Numeric	F0	SRP	2	Shift And Round
D2	MVC	1	Move	F1	MVO	2	Move With Offset
D3	MVZ	1	Move Zone	F2	PACK	2	Pack
D4	NC	1	AND	F3	UNPK	2	Unpack
D5	CLC	1	Compare Logical				
D6	OC	1	OR	F8	ZAP	2	Zero And Add
D7	XC	1	Exclusive OR	F9	CP	2	Compare
DC	TR	1	Translate	FA	AP	2	Add
DD	TRT	1	Translate And Test	FB	SP	2	Subtract
DE	ED	1	Edit	FC	MP	2	Multiply
DF	EDMK	1	Edit And Mark	FD	DP	2	Divide

Table 26. Typical SS-type instructions

ED, EDMK, SRP, and the last six instructions in the right-hand column operate on data stored in *packed decimal* format, which is different from the data formats used for the general register and floating-point instructions. We'll learn about them in Chapter VIII.

9.9. Writing SS-Type Instructions

Most SS-type instructions specify two addresses, and may have one or two length fields depending on whether you must specify the length of only one operand (type SS-1) or of both operands (type SS-2). Their formats are shown in Tables 27 and 29.

As with explicit and implied addresses, you can also specify explicit and implied *lengths* in SS-type instructions. When we use implied lengths the Assembler determines the values put into the length fields of the instruction, often by using the length attribute of a symbol. Implied lengths are *very* useful, and we'll see many examples.

This is the format of instructions with a single length field.

opcode	L_1	B_1	D_1	B_2	D_2

Table 27. Typical type SS-1 instruction with one length field

Addresses and lengths may be specified explicitly or implicitly, as summarized in the following tables. First, we examine the single-length instructions.

SS-1	Explicit Addresses	Implied Addresses
Explicit Length	$D_1(N_1,B_1),D_2(B_2)$	$S_1(N_1),S_2$
Implied Length	$D_1(,B_1),D_2(B_2)$	S_1,S_2

Table 28. Operands of type SS-1 single-length instructions

When you write an instruction with an explicit length, you provide a “Length Expression” or “program length”, denoted “N_1”. The Assembler generates object code with an “Encoded Length” or “machine length” denoted by “L_1”. This seems strange: why are they different?

The Assembler generates the value of L_1 by subtracting 1 from the value of N_1 (unless N_1 is zero). We'll see why this is done when we discuss SS-type instructions starting in Section 24.

Some examples of SS-type instructions with a single length field are:

```
MVC   0(80,4),40(9)       Explicit length and addresses
CLC   Name(24),RecName    Explicit length, implied addresses
TR    OutCh(,15),0(12)    Implied length, explicit addresses
XC    Count,Count         Implied length and addresses
```

where the symbol **OutCh** must be absolute. (This form is rarely used.)

SS-type instructions with two length fields have the format shown in Table 29.

opcode	L_1	L_2	B_1	D_1	B_2	D_2

Table 29. Typical type SS-2 instruction with two length fields

Many more combinations of explicit and implied lengths and addresses are available when you use SS-type instructions with two length fields. Some of the Assembler Language operand field combinations are shown below.

SS-2	Explicit Addresses	Implied Addresses
Explicit Lengths	$D_1(N_1,B_1),D_2(N_2,B_2)$	$S_1(N_1),S_2(N_2)$
Implied Lengths	$D_1(,B_1),D_2(,B_2)$	S_1,S_2

Table 30. Operands of type SS-2 two-length instructions

You can specify explicit lengths and addresses for either of the two operands; see Exercise 9.9.2.

As noted for SS-1 type instructions, the Encoded or machine lengths L_1 and L_2 are one less than the Length Expressions or program lengths N_1 and N_2. We'll see these again in Chapter VIII.

Some examples of SS-type instructions with two length fields are:

```
PACK  0(8,4),40(5,9)        Explicit lengths and addresses
ZAP   Sum(14),OldSum(4)     Explicit lengths, implied addresses
AP    Total(,15),Num(,12)   Implied lengths, explicit addresses
UNPK  String,Data           Implied lengths and addresses
```

The symbols **Total** and **Num** must be absolute for the third statement to be valid.

This SS-type instruction copies five bytes from a memory area named **AREA** to an area of memory named **FIELD**:

```
MVC   FIELD(5),AREA
```

Exercises

9.9.1.(2)+ The following operands could be used in SS-type instructions. State the operand for which they may be valid, for both SS-1-type and SS-2-type instructions, and whether a length is explicit or implied. (Validity and form may depend in the relocation attribute of the symbols.)

(1) `1(2)`
(2) `4(5,6)`
(3) `A(L'B)`
(4) `Line`

(5) `Line(80)`
(6) `XX(,5)`

9.9.2.(2)+ Make a table to show all possible combinations of explicit and implied addresses, and implicit and implied lengths, for SS-2 type instructions.

9.10. Summary

When describing the fields of both machine instructions and assembler instruction statements, we use notations like S_2, B_1, N, L_2, etc.

- Fields denoted S can be absolute *or* relocatable expressions, and are most often relocatable.
- Fields denoted B, D, I, L, N, and X must always be absolute expressions.

Terms and Definitions

Encoded Length
: The contents of a Length Specification Byte; one less than the value of the Length Expression (unless the Length Expression is zero, in which case the Encoded Length is also zero).

explicit address
: An address in which you specify the base register specification digit and the displacement as absolute expressions.

explicit length
: A length field that you specify explicitly.

implied address
: An address where you expect the Assembler to assign a base register specification digit and a displacement to an addressing halfword.

implied length
: A length field completed by the Assembler based on its analysis of the operand.

Length Expression
: A value you write in an SS-type instruction specifying the length of the operand(s).

machine length
: An Encoded Length.

mnemonic
: A character string representing an instruction, intended to be easier to remember than the *operation code* of the instruction.

opcode
: An abbreviation for *operation code*. Occasionally used when the term *mnemonic* is actually meant.

operation code
: The z/Architecture definition of an instruction's bit pattern to be decoded by the CPU to determine what actions it should take.

program length
: A Length Expression.

10. Establishing and Maintaining Addressability

```
    11        00000000
   111       0000000000
  1111      00        00
    11      00        00
    11      00        00
    11      00        00
    11      00        00
    11      00        00
    11      00        00
    11      00        00
1111111111   0000000000
1111111111    00000000
```

In Section 5 we saw how the CPU at *execution* time converts addressing halfwords into Effective Addresses. Now we will see how the Assembler derives addressing halfwords from the values of symbolic expressions at *assembly* time, and answer the question “How do we help the Assembler create addressing halfwords?”

This important information is provided in the USING assembler instruction statement.

10.1. The BASR Instruction

The RR-type Branch and Save (Register) instruction with mnemonic BASR is frequently used to generate a base address that provides addressability.[61] For now, we consider what happens when we write

```
BASR   R1,0
```

where the second operand register specification digit R_2 is zero. This instruction *when executed* replaces the contents of the general register specified by R_1 by the Instruction Address (IA) portion of the PSW. This address will necessarily be the address of the instruction *following* the BASR, because the IA was incremented by the BASR instruction's length (2 bytes) during the fetch portion of the instruction cycle.

In this RR-type instruction (unlike many other RR-type instructions), the zero second operand does *not* refer to general register zero! Instead, it means that only the described actions will occur *without* any “branch”, as the “Branch and Save” name implies. (We'll see in Chapter X that BASR is often used for branching, usually in subroutine linkages.)

Suppose the following short sequence of statements is part of a program that has been assembled and placed in memory to be executed. While we are giving the Assembler Language *statements* in Figure 35 on page 117, the assembled contents of memory will be hexadecimal machine language data, as shown in Figure 36 on page 118. Suppose the Program Loader has relocated the program so that the first instruction (the BASR) was placed at memory address X'5000'.

---

[61] The BASR instruction should be used in place of BALR in most situations; the main difference is that BALR inserts the ILC, CC, and Program Mask in the high-order 8 bits of the first operand register when executing in 24-bit addressing mode. BALR and BASR work the same way in 31-bit and 64-bit addressing modes.

```
Address   Name   Operation  Operand     Remarks

          *      Fragment of a simple program
  5000           BASR       6,0         Establish base address
  5002    BEGIN  L          2,N         Load contents of N into GR2
  5006           A          2,ONE       Add contents of ONE
  500A           ST         2,N         Store contents of GR2 into N
         --twenty-two (X'16') additional bytes of instructions, data, etc.--
  5024    N      DC         F'8'        Word integer 8
  5028    ONE    DC         F'1'        Word integer 1
```

Figure 35. A simple program segment

For this and the following examples, the instructions following the BASR are intended just to show how the Assembler creates addressing halfwords. Briefly, their actions are:

- L is the mnemonic for the RX-type (4-byte) machine instruction **L**oad. It copies the contents of a 4-byte (word) area of memory and puts it into a general register.
- A is the mnemonic for the RX-type (4-byte) machine instruction **A**dd. It adds a copy of the contents of a 4-byte (word) area of memory to the contents of a general register.
- ST is the mnemonic for the RX-type (4-byte) machine instruction **ST**ore. It replaces the contents of a 4-byte (word) area in memory with a copy of the contents of a general register.
- DC (**D**efine **C**onstant) is an Assembler instruction used to create constants. The two DC statements create word binary integers in memory.

The leftmost column in Figure 35 shows the memory address of each instruction and data item.

For now, we'll ignore what the instructions actually do, and focus on how they are assembled.

Exercises

10.1.1.(2) Use the lengths of the instructions and constants in Figure 35 to calculate their addresses in memory, and determine if the values in the figure are correct.

10.2. Computing Displacements

Now, suppose the program has begun execution. After the BASR has been executed, register 6 will contain X'00005002'. (Remember: BASR places the address of the *next* instruction into the register designated by the R_1 operand.) We can now use the address in register 6 as a *base address* for the instructions following the BASR, so the base register specification digit in subsequent addressing halfwords should be 6.

We can determine the proper displacement in the L instruction at X'5002' by using two important values: the known contents of register 6 (X'00005002') and the address of the word area named **N**. Using these values, we can now compute a displacement:

X'00005024' – X'00005002' = X'022'

Then, the assembled machine language instruction (using opcode X'58' for the mnemonic L) will be X'58206022'. When this instruction is executed, its Effective Address is

X'022' + X'00005002' = X'00005024',

the address of the word named **N** that we want!

If we continue this way for the rest of the statements, the "assembled" machine language instructions and data will give the desired results at execution time. That is, after program loading is complete, we want the memory areas starting at address X'5000' to contain the (hexadecimal) machine language data shown under "Assembled Contents" in Figure 36 on page 118.

```
Address   Assembled Contents    Original Statement

  5000    0D60                          BASR     6,0
  5002    58206022              BEGIN   L        2,N
  5006    5A206026                      A        2,ONE
  500A    50206022                      ST       2,N
                                ----------------------------------
  5024    00000008              N       DC       F'8'
  5028    00000001              ONE     DC       F'1'
```

Figure 36. Simple program segment with assembled contents

Remember that when the Assembler processes the BASR statement and produces two bytes of machine language code containing X'0D60', nothing is yet "in" register 6. It is only when this machine language instruction is finally *executed* by the processor that the desired base address will be placed in register 6.

So far, so good: we have constructed a sequence of instructions that will give a desired result if it is placed in memory at exactly the right place. You might ask "What would happen if the program is put elsewhere by the Program Loader?" So, let's suppose the same program segment begins at memory address X'84E8', as in Figure 37.

```
Address        Statement

  84E8                BASR     6,0
  84EA         BEGIN  L        2,N
  84EE                A        2,ONE
  84F2                ST       2,N
               --- the same 22 bytes of odds and ends ---
  850C         N      DC       F'8'
  8510         ONE    DC       F'1'
```

Figure 37. Same program segment, at different memory addresses

After executing the BASR, register 6 contains X'000084EA'. To address the contents of the word named **N** using register 6 as a base register, the necessary displacement is

X'0000850C' – X'000084EA' = X'022'

Similarly, the displacement necessary in the "A" instruction is

X'00008510' – X'000084EA' = X'026'

After completing the three addressing halfwords, the assembled machine language program would appear in memory as shown in Figure 38.

```
Address        Assembled Contents

  84E8         0D60
  84EA         58206022
  84EE         5A206026
  84F2         50206022
            -----------------
  850C         00000008
  8510         00000001
```

Figure 38. Same program segment, with assembled contents

The *identical* machine language program is generated in both Figures 36 and 38. We see that so long as the *same fixed relationship* is maintained among the various parts of the program segment (there are 22 bytes between the ST instruction and the word named N), the program segment could be placed *anywhere* in memory and still execute correctly. That is, the program is *relocatable*.

Indeed, we could have assumed that the program began at memory address zero (even though an actual program would not be placed there) because the contents of register 6 after the BASR is executed would be X'00000002', and the displacements would be calculated exactly as before.

10.3. Explicit Base and Displacement

Knowing what we need for the assembled program (the machine language instructions shown in Figures 36 and 38), we now write the instruction statements with *explicit* addresses in their second operands. Register 6 is the base register, and the displacements are those we just calculated. Then we can write the program as in Figure 39, using an assumed origin of zero for the LC. (Remember: we're describing *locations* at assembly time, not the execution time *addresses* we saw in the previous examples.)

```
Location     Name     Operation  Operand

  0000                BASR       6,0
  0002       BEGIN    L          2,X'022'(0,6)
  0006                A          2,X'026'(0,6)
  000A                ST         2,X'022'(0,6)
             --------- 22 bytes ----------
  0024       N        DC         F'8'
  0028       ONE      DC         F'1'
```

Figure 39. Program segment with pre-calculated explicit base and displacements

This example has two shortcomings. First, calculating displacements in advance is tedious (especially in large programs), and certainly error-prone. Second, if the relative positions of the parts of the program change in any way, we will be forced to recalculate some or all of the displacements.

Thus, our first simplification is to find a way to let the *Assembler* compute the displacements just as we did. Now, however, we can make good use of the values assigned by the Assembler to the symbols **BEGIN**, **N**, and **ONE**. (As noted in Section 7.6 on page 93, the values of the symbols are the values of the LC when the statement is processed.) Referring to Figure 39, the values assigned to the three symbols will be the value of the assumed origin plus X'0002', X'0024', and X'0028', respectively.

The key to this example is that when the program is *executing,* the base register (register 6) contains the address of the instruction named **BEGIN**. We use this observation to rewrite the program segment, as shown in Figure 40.

```
Location     Name     Operation  Operand

  0000                BASR       6,0
  0002       BEGIN    L          2,N-BEGIN(0,6)      (N-BEGIN = X'022')
  0006                A          2,ONE-BEGIN(0,6)  (ONE-BEGIN = X'026')
  000A                ST         2,N-BEGIN(0,6)      (N-BEGIN = X'022')
             ------- the usual 22 bytes -------
  0024       N        DC         F'8'
  0028       ONE      DC         F'1'
```

Figure 40. Program segment with explicit base and Assembler-calculated displacements

We have eliminated both of the shortcomings of the program segment in Figure 39: the displacements were not calculated in advance, and adding (say) four more bytes of instructions or data preceding the DC statements would not require the rest of the program to be rewritten. However, we have created another nuisance, since *every* instruction containing a reference to a symbol must now specify two extra items: the symbol **BEGIN** and the base register (6).

So, we need a way to make the Assembler do the rest of the work for us, after we have told it (1) which base register to use, and (2) the value that will be in it when the program is executed.

10.4. The USING Assembler Instruction and Implied Addresses

The USING assembler instruction provides exactly the information we need. It is written

```
         USING   base_location,base_register
```

where "base_location" is almost always a relocatable expression. (The base_location is sometimes called the "base", but it easy to mistake this for the "base_register".) The "base_register" operand is an absolute expression between 0 and 15, specifying the register to be used as a base register. (Zero is very rarely used.)

Thus, the statement

```
         USING  BEGIN,6
```

tells the Assembler to assume that register 6 may be used as a base register that at *execution* time will contain the relocated *address* of the instruction named by the symbol **BEGIN**. The Assembler can then calculate displacements relative to the *location* of **BEGIN**, and then use this assumption to create addressing halfwords with base register specification digit 6 and the calculated displacements.

We now rewrite the sample program segment of Figure 40 on page 119 to include the USING statement in Figure 41.

```
         BASR     6,0
         USING    BEGIN,6
BEGIN    L        2,N
         A        2,ONE
         ST       2,N
-----------------------
N        DC       F'8'
ONE      DC       F'1'
```

Figure 41. Program Segment with USING Instruction

If the initial LC value is zero, the value of the symbol **BEGIN** will be X'0002', and the values of the symbols **N** and **ONE** will be X'0024' and X'0028' respectively. To complete its derivation of the addressing halfword of the ST instruction, the Assembler needs only to calculate the difference between the location of the symbol **N** and the base_location of **BEGIN** specified in the USING instruction:

```
  X'0024' – X'0002' = X'022'
```

and this is the required displacement.

Similarly, the implied address of the operand **ONE** of the A instruction has value X'0028'; when the base_location value is subtracted, we find the displacement is X'026', as before. We say that the Assembler has *resolved* the implied addresses of the L, A, and ST instructions into base-displacement form. Thus, the machine language generated from this set of statements would appear exactly as in Figures 36 and 38. (Details about how the Assembler computes displacements and assigns base registers is described starting in Section 10.8.)

If the attempted calculation

```
  displacement = (operand value) – (base_location value)
```

yields a negative result or a value greater than 4095, the location referred to by the symbol is still not addressable with this base register, and some other solution is needed.[62]

---

[62] Section 20 describes long-displacement and relative-immediate instructions with a larger range of displacement values.

It is clear that the Assembler can make use of the information supplied by the USING statement *only for implied addresses.* If you provide an explicit base and displacement, the Assembler simply converts them to their proper binary form.

Two important features of the program segment in Figure 41 on page 120 should be noted.

1. The USING instruction does *absolutely nothing* about actually placing an address into a register; it merely tells the Assembler what to *assume* will be there when the program is executed.

   That is, your USING statement is a *promise* to the Assembler that if it computes displacements for you, everything will work properly when the program is executed. (It is very easy to mislead the Assembler, as we'll see in Section 10.11 on page 129.)

2. If the BASR instruction had been omitted, the contents of register 6 at execution time is probably unknown. There is no guarantee that correct Effective Addresses will be computed when the program is executed.

> **Remember!**
>
> A USING statement is your *assembly-time* promise to the Assembler that your program will obey that promise at *execution* time.

10.5. Location Counter Reference

The Assembler provides a convenient way to refer to the current value of the Location Counter, the *Location Counter Reference*. The term * in an expression has the current value of the LC, and is always relocatable.

We can rewrite the first two statements of our sample program as

```
BASR  6,0
USING *,6
```

with the same results as before. Remember that after the BASR instruction is assembled, the LC will have a value corresponding to the location of the next byte to be assembled. Because BASR will (at execution time) place the address of the following instruction into register 6, we can use a Location Counter Reference to specify the base_location, and not have to use a symbol (such as the symbol **BEGIN** in Figure 41 on page 120). to name the instruction following the BASR instruction.

A common technique for specifying base registers in a program is to choose a base register, write the statements

```
BASR  reg,0
USING *,reg
```

at the beginning of the program, and then carefully avoid modifying that register. For simple programs, specifying and using base registers is very easy.

It's important to remember that while the value of "*" changes as your program is assembled, the value used in the first operand of the USING statement does not: it has the value of the LC at the time the USING is processed by the Assembler.

Exercises

10.5.1.(2)+ A careless programmer inverted the order of his BASR and USING statements as follows:

```
USING *,12
BASR  12,0
```

Why is this wrong? What would you expect to happen?

10.6. Destroying Base Registers

Suppose an error was made in writing the statement with the L instruction, such that it became

```
BEGIN   L    6,N         Load contents of N into GR2
```

The comment in the remarks field is correct; the instruction is wrong, because the first operand was incorrectly written as 6 instead of 2.

The assembled program would then appear as in Figure 42.

```
Location   Assembled Contents    Statement

  0000     0D60                        BASR   6,0
                                       USING  BEGIN,6
  0002     58606022              BEGIN L      6,N     ←Wrong register!
  0006     5A206026                    A      2,ONE
  000A     50206022                    ST     2,N
                                 ---------------------------
  0024     00000008              N     DC     F'8'
  0028     00000001              ONE   DC     F'1'
```

Figure 42. Sample program segment with erroneous statement

This program would assemble correctly, since all quantities are properly specified. However, at *execution* time, things go wrong quickly.

Suppose again that the program is placed in memory by the Program Loader starting at address X'5000', so that when the L instruction is executed, register 6 contains X'00005002'. Now, the L instruction copies a word from memory at the address given by the second operand into the register specified by the first operand. However, the first operand in this case specifies register 6, instead of register 2 as intended. When the Effective Address of the operand named **N** is calculated during instruction *decoding,* register 6 contains the correct base address; but when the *execution* of the L instruction is complete, register register 6 will contain X'00000008' and *not* X'00005002', because the number at **N** was placed in register 6.

Now the fun begins. When the *next* instruction (A) is executed, the Effective Address calculated is

```
X'026' + X'00000008' = X'0000002E'
```

and not X'00005028', where the intended operand is found. In this case the Effective Address is not anywhere within the program, but is somewhere among the predefined fixed fields at the low end of memory; strange numbers will be added to register 2's initial (and unknown) contents. Finally, the ST instruction will attempt to store a word at X'0000002A', which should cause a storage protection exception. At this point, the program would stop.

This does not mean that if we accidentally destroy the contents of a base register, the CPU will be able to detect the error. (See Exercise 10.6.1.) It is partly a matter of chance how much damage such a program error can cause when the program is executed; indeed, when the CPU finally (if ever) detects an error, all evidence pointing to the offending instruction may have been lost, making error tracing difficult. (Register 6 may have been changed several times!) You must be very careful to guarantee the integrity of the contents of base registers.

Remember also that the Assembler makes no checks for instructions that might alter the contents of registers designated as base registers in USING statements.

Exercises

10.6.1.(3)+ In the erroneous program in Figure 42, consider the possibility that the word at N contained the decimal integer 20450. If the program began in memory at address X'5000', what would be in that area of memory after the ST instruction is executed?

10.7. Calculating Displacements: the Assembly Process, Pass One

Now, we'll examine more closely how the Assembler computes bases and displacements.

You can visualize assembly as making two *passes* over the program: that is, the Assembler "reads" the program twice. On the first pass, the Symbol Table is built; on the second pass, data in the Symbol Table is used to help generate the desired instructions and data.

First, you will remember that values are assigned to symbols by the Assembler as follows:

1. A statement is read and examined to determine its general character. It is also saved in a temporary place so it can be read again during the second pass over the program.
2. If the statement will generate instructions or data, the Assembler adjusts the Location Counter (if necessary) to satisfy alignment requirements, so that instructions begin on halfword boundaries, words begin on word boundaries, etc.
3. If a symbol appears in the name field of the statement, it is entered into the Assembler's *Symbol Table,* and (if it is not an EQU statement) is given the value of the Location Counter. That is, the symbol is *defined*, as described in Section 7.6 on page 93. (Of course, it will be an error if the symbol is already in the table with a value; this is called *multiple* or *duplicate* definition.)
4. The rest of the statement is scanned; if any other symbols are encountered, they are entered into the Symbol Table (if not there already), but numeric values are not assigned to their attributes. That is, if the symbol is not yet defined, it remains "undefined".
5. The length of the instruction or data to be generated from the statement is then added to the Location Counter. No data or instructions are generated at this time, however.

This process is repeated for each statement, until the end of the program is reached. Because the Assembler has made a complete scan or "pass" over the program's statements, this is called "Pass One" of the assembly. At this point the Symbol Table contains all the symbols in the program, whether or not they are defined.

The first assembly pass is sketched in Figure 43 on page 124, but the sketch is incomplete in many ways. For example, an EQU statement lets you assign a value to a symbol, and that value is taken from the expression in the operand field. Figure 43, however, only shows values being assigned to symbols using the Location Counter. It also omits any description of macro-instruction statements, and how symbols are treated in erroneous statements.

```
 ┌─────────────────┐
 │ Read statement  │
 │  and save it    │
 └────────┬────────┘
          │
     ┌────▼───┐ yes  ┌──────────┐
     │ END ?  ├─────►│to Pass 2 │
     └────┬───┘      └──────────┘
          │no
 yes ┌────▼─────┐
◄────┤ comment? │
     └────┬─────┘
          │no
 ┌────────▼───┐     ┌────────────┐     ┌─────────┐
 │Instruction ├────►│ symbol in  ├────►│is it in │
 │statement?  │yes  │name field? │yes  │sym-tbl? │
 └────┬───────┘     └──┬─────────┘     └─┬─────┬─┘
      │no            no│  ┌──────────┐ no│     │yes
      │                │  │ enter it │◄──┘     │
 ┌────▼────────┐       │  └────┬─────┘    ┌────▼─────┐
 │Undefined    │       │       │       no │does it   │
◄┤mnemonic:    │       │       ▼◄─────────┤have a    │
 │note error   │       │       │          │value?    │
 └─────────────┘       │  ┌────▼──────┐   └────┬─────┘
                       │  │set value  │     yes│
                       │◄─┤ from LC   │        │
                       │  └───────────┘   ┌────▼──────┐
 ┌───────────┐         │                  │note error │
 │enter in   │    ┌────▼───────────┐      └────┬──────┘
 │table,     │ Y  │symbol(s) in    │           │
 │no value   │◄───┤operand field?  │◄──────────┘
 └────┬──────┘    └────┬───────────┘
      │              no│
 ┌────▼────────────────▼──────────────┐
◄┤increment LC by instruction length  │
 └────────────────────────────────────┘
```

Figure 43. Sketch of pass one of an assembly

Exercises

10.7.1.(2) In the following program segment, resolve the implied addresses into base-displacement form, and fill in the four blank fields.

```
Loc   Object Code        Statement

5000 0DA0                       BASR  10,0
5002                            USING *,10
5002 41D0____                   LA    13,SAVE
5006 4110____                   LA    1,PARM
500A 4DE0____                   BAS   14,SUB
500E 50______                   ST    0,TBL(15)
                                - - -
512C                     SAVE   DS    18F
5174                     PARM   DC    A(TBL)
5178                     TBL    DS    10F
51A0                     SUB    STM   14,12,12(13)
```

10.8. Calculating Displacements: the Assembly Process, Pass Two

The Assembler now begins a second pass over the program by retrieving the statements from their temporary storage place. The Assembler creates machine language object code, converting instruction mnemonics to operation codes and using data in the Symbol Table to evaluate *all* expressions appearing in the statements.

The overall flow of the second pass of the assembly process is sketched in Figure 44. As noted following Figure 43 on page 124 describing the first pass of the assembly, this is a very abbreviated description, so don't attach great significance to the precise sequence of processing actions implied by the diagram.

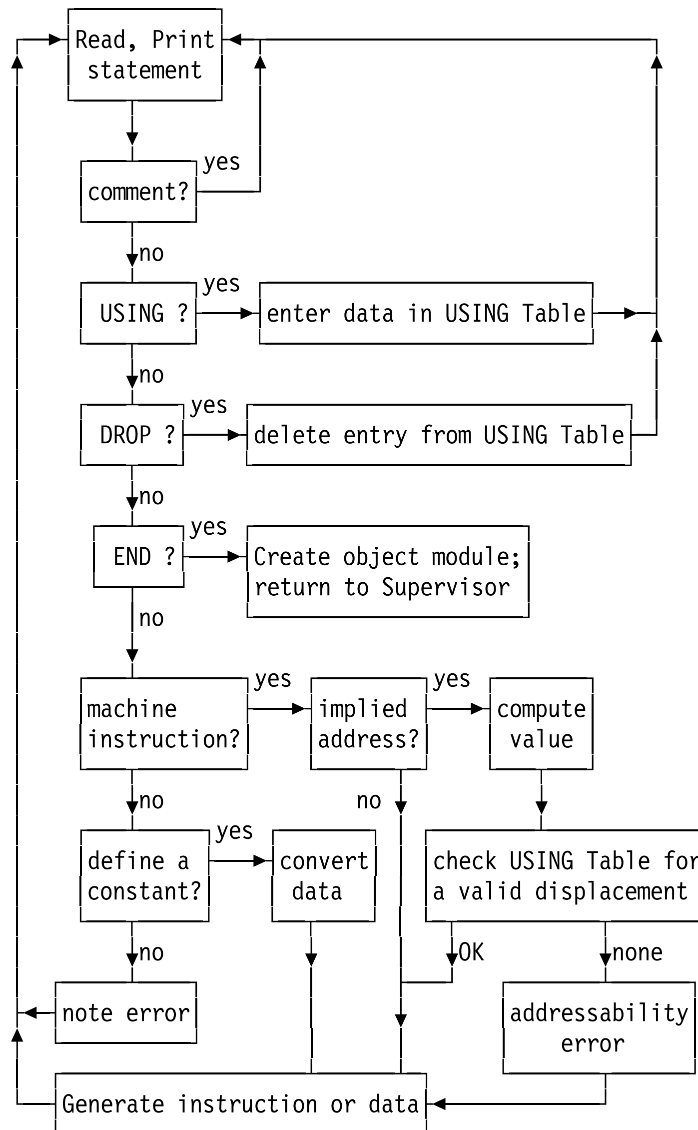


Figure 44. Sketch of pass two of an assembly

When a USING statement is encountered, the Assembler enters the value and relocation attributes of the first operand expression (the base_location), and the value of the second expression (the base_register number), into a *USING Table.*

Figure 45 on page 126 shows an example of a USING Table with one entry. The abbreviations "basereg" and "RA" denote respectively the base_register specified in the second operand of the USING statement, and the relocation attribute of the base_location expression from the first

operand of the USING statement. For now, the only importance of the relocation attribute is that it indicates whether the symbol is relocatable (RA=01) or absolute (RA=00).

basereg	base_location	RA
6	00000002	01

Figure 45. USING Table with one entry

When a subsequent instruction operand contains an *implied* address, the Assembler compares the value and relocation attribute of that expression to each entry in the USING Table. If a matching relocation attribute is found, and a valid displacement can be calculated from

```
displacement = (implied address value) – (base_location value)
```

then the Assembler inserts the computed displacement and the corresponding base_register digit into the addressing halfword of the instruction. The Assembler has *resolved* the implied address into base-displacement form, and the implied address is *addressable*.

For example, consider the second and third statements in Figure 41 on page 120. If the initial LC value assigned to the program was zero, the USING Table would contain an entry for register 6, with an associated relocatable base_location value of `X'00000002'`, the value of the symbol **BEGIN** illustrated in Figure 45.

When the third statement in Figure 41 on page 120 is processed, the value of the implied address is the value of the symbol **N**, or `X'00000024'`. The computed displacement is

```
X'00000024' – X'00000002' = X'022'
```

as we saw previously, so the completed addressing halfword is `X'6022'`.

Here is a way to summarize the description of operand address resolution: at assembly time, the Assembler computes a displacement:

```
displacement = (operand_location) – (base_location)
```

while at execution time, the CPU reverses this computation:

```
(operand address) = displacement + (base address)
```

Assembler-calculated displacements

The Assembler at assembly time does the reverse of what the CPU does at execution time.

It is important to give correct information in a USING statement because it specifies the intimate connection between the base_location at assembly time and the base address at execution time.

Remember that the difference between assembly-time locations and execution-time addresses in a relocatable program is only a single constant value,

Exercises

10.8.1.(2)+ In the blank fields provided in the six instructions below, show the values and addressing halfwords provided by the Assembler. Assume that the Location Counter values are as shown in the column headed “LOC”.

```
Loc   Object Code        Statement

10A20                           USING *,11
10A20 5830____                  L     3,X
10A24 4A30____                  AH    3,Y
10A28 10__                      LPR   4,3
10A2A 9034____                  STM   3,4,Z
10A2E 4240____                  STC   4,W
10A32 4770____                  BC    7,*+24
- - -                           - - -
10A76                    W      DS    X
10A78                    Z      DS    2F
10A80                    Y      DC    H'-72'
10A84                    X      DC    A(Z-W)
```

10.9. Multiple USING Table Entries

You can create more than one entry in the USING Table, so it is possible to have more than one valid resolution of an implied address into base-displacement form. Suppose we add another USING statement to the program, as in Figure 46:

```
Location    Name    Operation Operand     Remarks

  0000              BASR      6,0
                    USING     *,6         Original USING statement
  0002      BEGIN   L         2,N
                    USING     *,7         Added USING statement
  0006              A         2,ONE
  000A              ST        2,N
            ---------------------------
  0024      N       DC        F'8'
  0028      ONE     DC        F'1'
```

Figure 46. Program segment with second USING statement

For now, we ignore the fact that the contents of register 7 are unknown.

When the second USING is processed, the value of the Location Counter is X'00000006', so the Assembler makes a second entry in the USING Table, as shown in Figure 47.

basereg	base_location	RA
6	00000002	01
7	00000006	01

Figure 47. USING Table with multiple entries

When the *next* statement

```
         A      2,ONE
```

is processed, two possible valid resolutions are available for the implied address specified by the symbol **ONE**:

- If register 6 is used as a base register, the displacement is

  X'00000028' – X'00000002' = X'026'

  and the addressing halfword would be X'6026' (as in Figure 42 on page 122).

- If register 7 is used as a base register (again, ignoring the fact that its run-time contents are unknown), the Assembler determines that the displacement is

  `X'00000028' – X'00000006' = X'022'`

  and the addressing halfword would be `X'7022'`. (Similarly, the ST instruction could have an addressing halfword `X'701E'`.)

The Assembler must make a choice: which of the two valid resolutions should be selected for the completed machine language instruction?

The Assembler uses these resolution rules:

1. Find all USING table entries whose relocation attribute matches that of the implied address to be resolved.
2. Choose the base register that leads to the smallest displacement.
3. If more than one base register provides the same smallest displacement, choose the corresponding highest-numbered register.

Thus, the assembled program would appear as shown in Figure 48 below:

```
Location        Assembled Contents

 00000          0D60
 00002          58206022            Based on register 6
 00006          5A207022            Based on register 7
 0000A          5020701E            Based on register 7
                -----------------
 00024          00000008
 00028          00000001
```

Figure 48. Assembled contents when two USINGs are active

At this point, you could (correctly) observe that this program is seriously flawed, because the contents of GR7 at execution time could be "anything". When the A and ST instructions are executed, their operand addresses are likely to cause errors (whether or not they are detected immediately!).

The important lesson in this example is that the Assembler has no way of knowing that the information supplied in the statement

```
         USING  *,7
```

may not be valid. It can only trust that *you* have provided correct base_location and base_register data it can use to resolve implied addresses.

10.10. The DROP Assembler Instruction

It is also possible to *delete* entries from the USING Table. The DROP instruction tells the Assembler to remove the information corresponding to a given register. Its general form is

```
         DROP  register
```

where the "register" operand specifies the USING Table entry to be deleted.

For example, if the statement

```
         DROP  6
```

was inserted after the third statement, the L instruction named **BEGIN** in Figure 47 on page 127, the initial USING Table entry for register 6 would be deleted, and the USING Table would appear as in Figure 49 on page 129:

basereg	base_location	RA
	empty	
7	00000006	01

Figure 49. USING Table after DROP statement

Another form of the DROP statement is

```
         DROP
```

with *no* operand! This will cause all USING Table entries to be deleted. While this might seem odd, it's useful: if you have reached a part of your program where no valid base registers will be available at execution time, DROPping all the USINGs will avoid unexpected or unintended resolution of implied addresses in later parts of your program.

Exercises

10.10.1.(1)+ A frustrated programmer wrote the statements

```
DEAD     EQU   101
         DROP  DEAD
```

How would you expect the Assembler to deal with this impertinence?

10.10.2.(3)+ For each statement of the following program segment, show what will appear in the USING Table following each USING and DROP statement. Then, use that information to show the assembled machine language object code produced from the program segment. Assume the program segment begins at location X'4000'.

```
         BASR  9,0
         USING *,9
         L     4,*+54
         BASR  10,0
         USING *,10
         L     3,*+52
         DROP  9
         L     2,*+48
         DROP  10
         L     1,10(0,9)
```

What would be found in register 1 after the last instruction is executed? How does it depend on the address where the instructions are loaded into memory?

10.11. Addressability Errors

Addressability errors have many causes. These examples show some of the ways they can arise.

1. An operand value is larger than any USING Table base location value.

   ```
   BASR  6,0
   USING *,6
   L     2,*+5000
   ```

   Suppose the value of the Location Counter after the BASR instruction is X'002468'. This means that the value of the operand *+5000 is

   ```
   X'002468' + X'1388' = X'0037E0'
   ```

   and that the calculated displacement (for register 6) would be

   ```
   X'0037E0' – X'002468' = X'1388'
   ```

which is too large for a 12-bit displacement field. This means the operand is not addressable with 16-bit addressing halfwords.

2. An operand value is smaller than any USING Table base_location value. Again assuming the value of the LC after the BASR instruction is X'002468':

```
BASR  6,0
USING *,6
L     2,*-32
```

In this case the operand value is X'002448', leading to a negative calculated displacement, X'FFFFFFE0'. This means the operand is not addressable with 16-bit addressing halfwords.

3. The USING Table is empty. Suppose a second DROP statement is added after the A instruction in the program shown in Figure 46 on page 127, specifying register 7:

```
DROP  7
```

Then, the remaining entry in the USING Table would be deleted, and the USING table would appear as in Figure 50 below.

basereg	base_location	RA
	empty	
	empty	

Figure 50. USING Table after second DROP statement

Because there are no entries left in the USING Table, there is no way for the Assembler to resolve the implied addresses of any following instructions, and an addressability error would be noted for those statements.

Exercises

10.11.1.(3)+ Suppose these instructions are assembled and then executed in a program:

```
B        BASR  6,0
         USING *,6
         L     2,B
```

What (if anything) would you expect to appear in GR2?

10.12. Resolutions With Register Zero (*)

Although USING statements specifying absolute base_locations are rare, they are allowed; absolute implied address expressions follow the same resolution rules as relocatable expressions. In most cases, there is no entry in the USING Table with an absolute base address, so the Assembler proceeds as though a *hidden* or *implied*

```
USING 0,0                Assembler's implicit USING
```

is always present. You can think of the USING Table appearing like this:

basereg	base_location	RA	
0	00000000	00	Assembler's hidden USING-Table entry
–	etc.	—	

Thus, an *implied* address such as

```
         LA    3,1000         Implied address = 1000 = X'3E8'
```

would be resolved to the addressing halfword X'03E8', with base register zero.

In the example in Figure 34 on page 109, we saw an instruction with an absolute implied S_2 operand:

```
43000468             IC   0,1128
```

The generated object code shows that the second operand was resolved with base register zero.

Now, suppose you wrote a USING statement with an absolute base address:

```
         USING 400,9          Base Address = 400 = X'190'
         LA    3,1000         Implied address = 1000 = X'3E8'
```

so the USING Table would look like this:

basereg	base_location	RA
0	00000000	00
9	00000190	00
–	etc.	—

The Assembler follows its usual resolution rules, and finds that there are *two* valid resolutions with addressing halfwords X'03E8' and X'9258'. Since the latter provides the smallest displacement, the Assembler chooses the resolution with base register 9! Fortunately, the Assembler will issue a diagnostic message whenever a USING with an absolute operand appears to overlap with its implicit USING 0,0 statement.

If the original resolution using base register zero is required no matter what other USINGs are active, the operand should be written explicitly, as

```
         LA    3,1000(0,0)   Explicit displacement=1000, base=index=0
```

Thus, we add one further resolution rule when absolute implied addresses have not been resolved according to the three previous rules:

4. If no previous resolution has been completed, *and* the implied operand is absolute *and* has value between 0 and 4095, use General Register 0 as the base register and the value of the implied address expression as the displacement.

This behavior is used often in Assembler Language programs. If any implied address has absolute nonnegative value, a valid displacement can always be computed only if that value does not exceed 4095.[63]

According to the rules for evaluating expressions, attempting to compute a displacement for a relocatable symbol using an absolute base_location would require that the displacement be relocatable, which is invalid. That is, a valid displacement cannot be calculated from

```
(absolute) displacement = (relocatable operand) – (absolute base_location) (??)
```

Similarly, an absolute implied address cannot be resolved into base-displacement form using a register whose base_location is relocatable, since a valid displacement cannot be computed from

```
(absolute) displacement = (absolute base_location) – (relocatable operand) (??)
```

[63] Section 20 shows how to use a much larger range of displacement values with long-displacement instructions.

It is possible (but *not* recommended!) to specify USING statements with register zero as the base register,[64] but the Assembler will always assign a base address of zero to register zero.

Exercises

10.12.1.(1)+ The Assembler tries to resolve absolute implied addresses into an addressing halfword containing a zero base digit, and a displacement of the value of the implied address. Do you think this is desirable? Would you prefer that the Assembler diagnose absolute implied addresses as an error?

10.13. Summary

In summary, the ordinary USING statement provides two major features:

1. A *base_location* relative to which the Assembler can calculate displacements.
2. A *base_register* to be used in addressing halfwords of implied addresses whose displacements were calculated as being addressable with this register.

The information conveyed in a USING statement is only, and no more than, a promise that you make to the Assembler. You are asserting that if it uses the base_location and base_register specified in your USING statement to calculate addressing halfwords at assembly time, then the CPU will calculate correct Effective Addresses at execution time.

The rules for resolving implied addresses into base-displacement form can be difficult to remember, and forgetting them can sometimes lead to programming errors that are difficult to correct.[65]

USING Resolution Rules

1. The Assembler searches the USING Table for entries with a relocation attribute matching that of the implied address (which will almost always be simply relocatable, but may be absolute).
2. For all matching entries, the Assembler checks to see if a valid displacement can be derived. If so, it will select as a base register the register that yields the smallest displacement.
3. If more than one register yields the same smallest displacement, the Assembler will select the highest-numbered register as a base register.
4. If no resolution has been completed, and the implied address is absolute, try a resolution with register zero and base zero.

A minor addition to these rules will apply when we discuss instructions with long 20-bit signed displacements in Section 20.

The relocatability attribute of any given symbol almost always has a single value; it won't matter if we ignore “complex relocatability” situations for now, because they don't affect addressability. However, it is not unusual for programs to use many *different* relocatability attributes to correctly describe its symbols.

In Chapter XI we will see powerful extensions to the USING statement — *Labeled* and *Dependent* USINGs — that give you much greater control over USING resolutions.

---

[64] When we discuss Dummy Control Sections in Section 39, we will see that there can be times when specifying a zero base register is a reasonable practice.

[65] Some programmers note that “USING” is part of “confusing”.

10.13.1. How the Assembler Helps

The Assembler simplifies many programming tasks:

1. It automatically resolves addresses into the base-displacement and other forms used by System z. The Assembler determines the needed base and displacement so that correct Effective Addresses will be computed at execution time.
2. Rather than remembering that operation code `X'43'` places a byte from memory into the right end of a general register, a *mnemonic operation code* IC ("Insert Character") gives a simple indication of what the operation code does.
3. Symbols let you name areas of memory and other objects in your program.
4. Diagnostic messages help you find possible errors and oversights.
5. The Assembler converts data from convenient external representations into internal forms.
6. It creates relocatable object code to be combined with other programs by the linker.
7. It provides lots of other helpful information such as symbol and register cross-references.
8. Using macro-instructions, you can define your own instruction names to supplement existing instructions, and your macro instructions can make use of previously defined sequences of statements, including other macros!
9. The High Level Assembler provides an optional summary of all USING Table activity, in the form of a USING Map. If you specify `USING(MAP)` as part of the parameter string when you invoke the High Level Assembler, it will display all USING and DROP activity for the entire program.

Exercises

10.13.1.(3) Some older assemblers let you redefine symbols in EQU statements. Thus, you could write

```
A        Equ   6                 Define a value for A
         - - -                   Write statements using A's value
A        Equ   32                Define a new value for A
         - - -                   Statements using A's new value
```

How would the assembler's treatment of the Symbol Table be changed? What would happen if *any* symbol could be redefined?

Terms and Definitions

addressability
The ability of the Assembler to calculate a displacement and assign a base register to an implicit addressing expression, using information in the USING Table.

addressability error
The inability of the Assembler to derive an addressing halfword for an implicit operand.

base_location
The first operand of a USING instruction at assembly time.

base_register
The second operand of a USING instruction at assembly time.

DROP assembler instruction
An instruction telling the Assembler to eliminate one or more entries from its USING Table.

Symbol Table
A table used by the Assembler to hold the names, values, and attributes of all symbols in a program.

USING statement
A promise to the Assembler that addressing halfwords can be derived correctly from the base_location and base address information provided in the instruction.

USING Table
An internal table used by the Assembler to hold information provided in USING instructions.

Programming Problems

Problem 10.1.(1) Write and assemble a program segment like the one in Figure 41 on page 120, with the following additional statements:

1. Following the last DC statement, place an Assembler instruction statement with the mnemonic END in the operation field.
2. Replace the dotted line that means “twenty-two additional bytes” with an Assembler instruction statement with DS in the operation field and 22X in the operand field.
3. Preceding the first statement place an Assembler instruction statement with the mnemonic START in the operation field, and `X'5000'` in the operand field.

Assemble the program, and save the Assembler's listing. Then, replace the `X'5000'` operand in the START statement with the `X'84E8'`, and re-assemble the program, saving the second listing. Verify that the assembled machine language program is the same in both listings, and that the same bases and displacements are calculated by the Assembler for all instructions that require them. If time and budget permit, do the same for the programs in Figures 39 and 40.

Chapter IV: Defining Constants and Storage Areas

```
IIIIIIIIII  VV        VV
IIIIIIIIII  VV        VV
    II      VV        VV
    II      VV        VV
    II      VV        VV
    II      VV        VV
    II      VV        VV
    II       VV      VV
    II        VV    VV
    II         VV  VV
IIIIIIIIII      VVVV
IIIIIIIIII       VV
```

The three sections of this chapter treat the DC (Define Constant) and DS (Define Storage) assembler instruction statements, and methods used to define data and storage areas in Assembler Language programs.

- Section 11 describes the Assembler's basic data definition instruction, DC.
- Section 12 discusses the most often-used data types, introduces the powerful constant-referencing mechanism provided by literals, and the LTORG instruction to control their location in your program.
- Section 13 demonstrates methods for defining and describing data areas in ways that simplify data manipulation problems, including the very useful DS, EQU, and ORG instructions.

11. Defining Constants

```
    11          11
   111         111
  1111        1111
    11          11
    11          11
    11          11
    11          11
    11          11
    11          11
    11          11
1111111111  1111111111
1111111111  1111111111
```

In the preceding sections we used the DC assembler instruction to create constants in the program. Now we'll describe basic rules for defining constants of any type.

System z supports a very rich variety of data types, and various lengths and precisions can be specified for most of them. Among the "native" data types the Assembler supports are:

1. Fixed-point data (two's complement binary), signed and unsigned
   - doubleword precision (64 bits)
   - word precision (32 bits)
   - halfword precision (16 bits)
   - byte precision (8 bits)
2. Logical data (binary and hexadecimal)
   - doubleword (64 bits)
   - word (32 bits)
   - one byte (8 bits)
   - varying-length (1 to 256 bytes)
3. Address-valued (3, 4, and 8 bytes)
4. Character data (1 to 256 bytes) in EBCDIC, Graphic (Double-Byte), ASCII, and Unicode formats.[66]
5. Decimal data (sign-magnitude representation)
   - zoned decimal (1 to 16 digits)
   - packed decimal data (1 to 31 digits)
6. Floating-point data (sign-magnitude representation in binary, hexadecimal, and decimal formats)
   - short precision (4 bytes)
   - long precision (8 bytes)
   - extended precision (16 bytes)

[66] We'll investigate some non-EBCDIC character data types in Section 27.

Data for each of these types is defined using the DC ("Define Constant") assembler instruction, with many options for each type.

Be Careful!

The DC instruction doesn't really define an unchangeable *constant* value, because you can change it at execution time. (It's only constant if you don't change it!) The instruction might better be called "Define Data with Initial Value". We'll see that literals can help you define what appear to be "true constants" In Section 13.9 on page 174.

You will usually write values in data definitions in the *external* representation most convenient for you. The Assembler then converts the data into the *internal* form used by your program, the CPU, and other devices.

As indicated in previous examples, a DC assembler instruction statement may have name, operation, operand, and remarks field entries; the operation and operand field entries are required.

11.1. Defining Constants

We'll start with the F-type constant we saw in several earlier examples. The assembler instruction statement

```
DC    F'8'
```

creates a word binary integer constant (X'00000008'), placed on a word boundary. In this statement, four items were specified or implied:

1. The *type* of desired conversion from the external form you wrote in the statement, to an internal representation. For type F, the decimal value is converted to a two's complement binary integer.
2. The *nominal value* of the constant, the decimal value 8.
3. The *length* of the constant, which for type F is implicitly four bytes.
4. The *alignment* in memory of the constant, implicitly on a word boundary for type F.

Some other types of conversion, and the letters that specify the types, are character (C), binary (B), hexadecimal (X), halfword binary integer (H), and address constant (A and Y). Here are examples of some of these types:

```
DC    H'8'          halfword binary integer
DC    C'/'          character constant
DC    X'61'         hexadecimal constant
DC    B'01100001'   binary constant
```

The last three constants are each one byte long, and contain identical bit patterns.

Important to remember

The binary, character, and hexadecimal self-defining terms use the same notation as constants of those types. It can be easy to forget that a self-defining term is just a number, while the operand of a DC statement defines an initial value in storage.

Exercises

11.1.1.(1) Constants of types B, C, and X are written in a form very much like self-defining terms of the same types, as in

```
DC    B'11010001',C'J',X'C5'
```

Constants with *decimal* values are written as (for example) F-type constants, as in

```
         DC    F'8'
```

Why do you think the designers of the Assembler Language made this choice, rather than allowing you to write this constant in the simpler form

```
         DC    8  ?          Alternative to F'8' ?
```

11.2. DC Instruction Statements and Operands

The operand field entry contains one or more operands separated by commas. An operand of a DC statement has four parts, with no spaces between them:

1. a duplication factor (if omitted, it defaults to 1)
2. a letter (or pair of letters[67]) specifying the type of representation
3. zero to four modifiers
4. the *nominal value* of the constant, enclosed in a pair of delimiters. The delimiters are either apostrophes or parentheses, depending on the type of the constant.

Of these four parts, only the second (the type) and fourth (the nominal value) are required. In the example above, **F'8'** specifies type **F** and nominal value **8**.

The three important modifier types are length, scale, and exponent.[68] Only length will be discussed here.

DC Operands

This may help you remember the order of of the fields: duplication factor (*d*), type (T), modifiers (*m*), and nominal value (V), where the required type and value are specified in capital letters: *d*T*m*V

The nominal value part of the operand is specified in different ways for different constant types. For F-type constants, the value is written as a string of *decimal* digits, preceded by an optional + or − sign and followed by an optional decimal exponent. For B-type constants, the value is expressed as a string of *binary* digits, so F'110' and B'110' are quite different.

The constant type also determines what *conversion* from external to internal representation should be performed: the internal representations of F'110' (binary word), X'110' (hexadecimal constant), E'110' (short floating-point), Z'110' (zoned decimal), and P'110' (packed decimal) are different, even though they all have the same nominal value.

11.2.1. Blanks in Nominal Values

Some constant types delimited by apostrophes (like F'8') let you put blank spaces between the digits to improve readability. For example, you can write either

```
         DC    F'12345678'
or
         DC    F'12 345 678'
```

We'll see more examples as we investigate various data types.

[67] We'll discuss type extensions in Section 12.8.

[68] HLASM supports another constant modifier and attribute, “Program”. It is used almost entirely in conditional assembly macro-instruction statements.

11.3. Boundary Alignment

Many constant types have “natural” boundary alignments. For example, the F-type constant is naturally word-aligned. Other constant types don't have a natural alignment; Table 32 on page 153 (Section 12.5) and Table 33 on page 158 (Section 12.8) summarize default alignments for many common data types.

There is an important relationship between boundary alignment and the presence of a byte-length modifier, which helps you align constants and data properly.[69] This will be discussed shortly, in Section 11.4.

By default, the Assembler initializes the Location Counter to zero. If you specify an initial LC value at the start of the program, the Assembler rounds it up (if necessary) to a multiple of eight to ensure that the program begins on a doubleword boundary.[70] Then, if a constant must fall on a specific boundary, the Assembler only needs to be sure that the Location Counter is divisible by the proper power of two (such as 2, 4, or 8) at the location of the leftmost byte of the constant.

The Linker and Program Loader respect this assumed alignment for the beginning of the program. This guarantees that data and instructions will be aligned on the desired boundaries when the program is loaded into memory for execution.

Suppose that after a sequence of instructions has been processed, the value of the LC is `X'00012E'` (on a halfword boundary). If another machine instruction is assembled at this point, it would begin on this halfword boundary between two word boundaries. But if the next statement is instead

```
DC    F'8'
```

the Assembler must place it on a word boundary to force the desired alignment.

Generating the four bytes of this constant beginning at the halfword-aligned location `X'00012E'` could be incorrect, because instructions referring to word constants normally expect the address to be on a word boundary. To avoid alignment errors, the Assembler automatically *skips* enough bytes to obtain the desired alignment. The LC would be increased to `X'000130'` (now word-aligned) before the word constant is assembled. The LC has value `X'000134'` after the constant is processed; it would be `X'000132'` if automatic alignment was not done.

Automatic alignment is *not* performed (bytes are not automatically skipped) if:

1. it isn't needed: that is, the LC happens to fall on the desired boundary; or
2. the type of constant specified doesn't require alignment, such as types C, B, or X (among others); or
3. a length modifier is present.

You can tell the Assembler to do no boundary alignment even if the constant type normally requires it.[71]

---

69 We'll see in Section 17.5 that constants can also have bit-length modifiers, but here we use the term “length modifier” to mean “byte length modifier”.

70 HLASM provides the SECTALGN option to let you specify even more restrictive boundaries. See the *High Level Assembler Programmer's Guide* for details.

71 For details, consult the *High Level Assembler Programmer's Guide* for the NOALIGN option. However, few programs use this option.

11.4. Length Modifiers

Length modifiers let you specify (within limits) a constant's exact length in bytes.[72] When used, we say that an *explicit length* was specified.

A length modifier is written immediately following the letter specifying the data type, in the form

```
Ln     or     L(expr)
```

The quantity “n” is an unsigned, nonzero decimal self-defining term, and “expr” is a positive absolute expression enclosed in parentheses. The length modifier specifies the constant's length. Any symbols appearing in the length modifier expression must be defined before they are used in the length modifier expression, so that it can be evaluated immediately.[73] For example, the statements

```
         DC    FL3'8'
```

and

```
         DC    FL(2*4-5)'8'
```

both cause the three-byte constant X'000008' to be assembled at the location specified by the Location Counter; no boundary alignment is performed. In practice, length modifiers are used mostly with constants of types C and X, and very rarely with type F and other normally-aligned constants.

Because alignment is automatic *only*

(1) when the length is *implied* (that is, when no length modifier is given), and

(2) for constant types for which alignment is the default action,

the two statements

```
         DC    F'8'
```

and

```
         DC    FL4'8'
```

define the same constant, but the first is automatically aligned and the second is not.

When a symbol appears in the name field of a DC assembler instruction statement, boundary alignment affects the symbol's value. Suppose the value of the LC is X'00012E' when each of the statements in Figure 51 is encountered.

```
Explicit DC    FL4'8'         Explicit length = 4 bytes, not aligned
Implied  DC    F'8'           Implied length = 4 bytes, word aligned
```

Figure 51. Implied and explicit length specifications

Because no boundary alignment is performed for the first constant, the value of the symbol **Explicit** will be X'00012E'. For the second constant, two bytes must be skipped to achieve the required word alignment. If we refer to the constant using the symbol **Implied**, the symbol will have the value of the location of the first byte of the constant, X'000130'.

Symbol definition

When a symbol is defined, it is given its value *after* bytes are skipped for boundary alignment.

---

[72] It is also possible to specify a constant's length in *bits*, using a bit-length modifier. They have specialized uses; we will describe them in Section 17.5 on page 257.

[73] Sometimes the Assembler will let you define symbols after they are used in length modifier expressions, but it's safest to make sure they're defined before they're used in length modifiers.

As a general rule, the Assembler never automatically assigns the location of skipped bytes as the value of a symbol.[74] This includes cases where a byte must be skipped to ensure that an *instruction* begins on a halfword boundary. When bytes are skipped to achieve alignment of a following constant or instruction, the Assembler will insert bytes containing all zero bits into the bytes skipped.

Proper boundary alignments can be important: some instructions require aligned operands. Also, operand misalignment can affect the performance of your applications, because the CPU may need to bring more data from memory than your instruction actually requires.

Exercises

11.4.1.(2) What data is generated by these constants?

```
(1)      DC    FL1'-127'
(2)      DC    FL2'+128'
(3)      DC    FL3'-99,+99'
(4)      DC    FL1'+127'
```

11.4.2.(1)+ For these constants:

```
(1)      DC    F'11',FL3'12',FL3'13'
(2)      DC    F'21',FL2'22',FL2'23'
(3)      DC    F'31',FL4'32',FL3'33'
```

on what boundaries are the constants 13, 23, and 33 aligned?

11.5. Duplication Factors and Multiple Operands

A duplication factor (sometimes called a multiplicity, replication, or repetition factor) specifies the number of times the constant or constants in the operand will be duplicated; it is written immediately *preceding* the letter specifying the constant type. It may be either an unsigned decimal self-defining term, or a nonnegative absolute expression enclosed in parentheses. Any symbols appearing in the duplication factor expression must be defined prior to their use in the duplication factor.[75] For example, both

```
Three8s  DC    3F'8'              Duplication factor 3
```

and

```
Three8s  DC    (5/2+1)F'8'
```

are equivalent to writing the three statements

```
Three8s  DC    F'8'               Three statements
         DC    F'8'
         DC    F'8'
```

You can write more than one operand in the operand field entry of a DC instruction, so you will get the same result by writing

```
Three8s  DC    F'8',F'8',F'8'     Three operands
```

Duplication factors apply only to *operands*, not to statements.

For example, if you write

```
         DC    F'7',2F'4',3F'9'
```

[74] You can find ways to do it if you like, but there's no real value in doing so. (Why refer to something so uninteresting?)

[75] Sometimes the Assembler will let you define symbols after they are used in duplication factor expressions, but it's safest to make sure they're defined before they're used in duplication factors.

the Assembler generates six word-length, word-aligned constants: one with value 7, two with value 4, and three with value 9.

There are occasionally important uses for DC statement operands with a *zero* duplication factor. In such a case, the Assembler first skips as many bytes as necessary to properly align the constant specified by the operand, and then generates *no* data for that operand. This means that the Location Counter is not further incremented for that operand, after alignment (if any). Thus we could generate a word-aligned 4-byte constant with a statement like

```
         DC    0F'0',FL4'-1'
```

or even

```
         DC    0F,FL4'-1'
```

Zero duplication factors are discussed further in Section 13.2 on page 160.

11.6. Multiple Nominal Values

For almost all constant types, the nominal value may actually be a *sequence* of values separated by commas, as in

```
Three8s  DC    F'8,8,8'          One operand, 3 nominal values
```

This is equivalent to

```
Three8s  DC    3F'8'             One operand, duplication factor 3
```

and

```
Three8s  DC    F'8',F'8',F'8'    Three operands
```

Which format you use is largely a matter of taste and convenience. For example, you could specify a table of constants with a statement such as:

```
TABLE    DC    F'1,2,3,4,5,6,7,8,9,10'
```

Figure 52. Multiple constants

Each generated constant is a word integer, aligned on a word boundary.

In cases where multiple constants are specified, any symbol in the name field (in this example, **TABLE**) is given the value and Length Attribute associated with the *first* constant generated.

Exercises

11.6.1.(2) A meticulous programmer determined that 10^9 is the largest power of ten that will fit in a word binary integer, and wanted to define a constant of that value. To ensure that he wrote the constant with the correct number of zeros, he wrote the statement

```
TEN_to_9 DC    F'1,000,000,000'
```

What would be generated? What would you recommend?

11.6.2.(1) What will be generated by this constant?

```
         DC    2F'1,-1'
```

11.7. Length Attributes

Although its many benefits will become clear later, we noted in Section 7.3 on page 89 that the *Length Attribute* of a symbol can be very useful. Its value is determined by the statement in which the symbol is defined.

1. The Length Attribute of a symbol naming an instruction is the length of the instruction. Thus, the Length Attribute of the symbol **LOAD** in

   ```
   LOAD     LR    R7,R3        (from Example 8_4_1 on page 100 in Section 8.4.)
   ```

   is 2, and the Length Attribute of the symbol **BEGIN** in

   ```
   BEGIN    L     2,N          (from Figure 35 on page 117)
   ```

   is 4.

2. If a symbol is the name of a DC statement, its Length Attribute is the length of the first generated constant, *ignoring* duplication factors. Explicit lengths and Length Attributes may be assigned with a length modifier; otherwise the Length Attribute is the implied length. Thus, the three symbols in

   ```
   Implied  DC    F'8'         (from Figure 51 on page 140)
   Explicit DC    FL4'8'
   Three8s  DC    3F'8'
   ```

   all have Length Attribute 4.

3. If the symbol names a DC statement whose first operand contains multiple values, the symbol's Length Attribute is the length of the first generated constant, as noted for the symbol **Three8s** above. Similarly, the Length Attribute of the symbol **TABLE** in

   ```
   TABLE    DC    F'1,2,3,4,5,6,7,8,9,10'  (from Figure 52 on page 142)
   ```

   is 4, even though the statement defines constants occupying 40 bytes.

4. If the symbol names a DC statement with more than one operand, the Length Attribute assigned to the symbol is determined from the first operand only, according to the previous rules. Thus,

   ```
   TwoCons  DC    F'2',FL2'-2'
   ```

   would assign 4 as the Length Attribute of **TwoCons**.

5. A symbol defined in an EQU statement to have the value of a self-defining term is assigned a Length Attribute of 1. Thus, the symbol **ZILCH** in

   ```
   ZILCH    Equ   7           (from Example 8_4_2 on page 100 in Section 8.4.)
   ```

   has Length Attribute 1. (The EQU assembler instruction is described further in Section 13.3 on page 162.)

11.8. Decimal Exponents (*)

Some numeric constants can be simplified by using either a *decimal exponent* or an *exponent modifier*. When you want to generate a constant with several trailing zeros, both forms let you omit the trailing zeros.

11.8.1. Decimal Exponents

A decimal exponent is written as part of the nominal value of the constant. Following the numeric portion, write the letter E followed by a signed or unsigned integer. For example:

```
F100A    DC    F'1E2'            Generates X'00000064'
F100B    DC    F'1000E-1'        Generates X'00000064'
F1000    DC    F'1E3'            Generates X'000003E8'
FBillion DC    F'1E9'            Generates X'3B9ACA00'
```

11.8.2. Exponent Modifiers

An exponent modifier is written following the constant's type, and following any other modifiers. For example:

```
F100A    DC    FE2'1'            Generates X'00000064', aligned
F100B    DC    FE-1'1000'        Generates X'00000064', aligned
F100C    DC    FL4E2'1'          Generates X'00000064', unaligned
F1000A   DC    FE3'1'            Generates X'000003E8', aligned
F1000B   DC    FL3E3'1'          Generates X'0003E8',   unaligned
FBillion DC    FE9'1'            Generates X'3B9ACA00', aligned
```

You can write constants with both an exponent modifier *and* a decimal exponent in the nominal value; the power of 10 applied to the numeric portion of the nominal value is the sum of the two. For example:

```
F100A    DC    FE1'1E1'          Generates X'00000064'
F100B    DC    FE-1'1E3'         Generates X'00000064'
F1000A   DC    FE-7'1E10'        Generates X'000003E8'
FBillion DC    FE5'1E4'          Generates X'3B9ACA00'
```

We'll see more about scale and exponent modifiers in Section 32.3, on page 574.

Exercises

11.8.1.(2) Show the constants generated by these statements, indicating which are aligned by default and which are not.

```
(1)       DC    FE1'2E3'
(2)       DC    FE-1'5E5'
(3)       DC    FL2E2'1E2'
(4)       DC    FL4'8E1'
```

11.8.2.(1) Rewrite the intended constant in Exercise 11.6.1 on page 142 using (a) a decimal exponent and (b) an exponent modifier.

11.8.3.(2)+ In following program segment, determine the values assigned to the Location Counter in the last four statements. Then complete the “Object Code” column for the four statements with spaces provided.

```
Loc    Object Code        Statement

000000                    Ex11_8_3 START     0
000000 05F0                        BALR      15,0
000002                             USING     *,15
000002 __________                  L         2,X
000006 __________                  A         2,Y
00000A __________                  S         2,Z
00000E __________                  ST        2,RESULT
000012                             PRINTOUT  RESULT,*
000038                    DUMMY    DC        0CL16'GARBAGE'
______                    X        DC        F'2'
______                    Y        DC        F'15'
______                    Z        DC        F'3'
______                    RESULT   DC        F'0'
```

Terms and Definitions

boundary alignment
The Assembler's action in incrementing the Location Counter so that its value is adjusted to the boundary required by an instruction or by a constant operand.

constant type
A letter specifying the internal data representation for a generated constant.

decimal exponent
A letter E attached at the end of the digits of a numeric constant, followed by a positive or negative integer giving the power of ten by which the value of the digits is multiplied.

duplication factor
The number of times a constant operand should be assembled.

exponent modifier
A modifier specifying a positive or negative power of ten to be multiplied by the nominal value of a constant.

length modifier
A modifier specifying the exact length to be used for a constant, rather than its default length.

modifier
A value following the constant type, specifying other information about the constant's Length, Scale, and Exponent.

nominal value
The value you write between delimiters or value separators to specify the assembled value of a constant.

12. Basic Constants

```
   11       2222222222
  111      222222222222
 1111      22        22
   11                22
   11                22
   11               22
   11             22
   11           22
   11         22
   11       22
1111111111 222222222222
1111111111 222222222222
```

We now use the general rules of the previous section to describe seven basic constant types used in many programs — F, H, A, Y, C, B, and X — and the useful form of constants called “literals”.

12.1. F-Type and H-Type Constants

We saw the F-type constant in earlier examples, so we will just summarize its properties here. The implied length is four, and the default alignment is to a word boundary. If an explicit length is specified, no alignment is performed and the length may be between 1 and 8 bytes. The nominal value of the constant is an optionally signed string of decimal digits. Thus, you can write

```
         DC    FL1'-10'      Generates X'F6', not aligned
         DC    FL8'-10'      Generates X'FFFFFFFFFFFFFFF6', not aligned
```

The H-type constant is similar to type F, specifying two's complement binary integer conversion to a 16-bit integer in two bytes aligned on a halfword boundary. Thus

```
         DC    H'-10'
```

places the constant X'FFF6' on the next available *halfword* boundary. If an explicit length is given, there is no difference between constants of types F and H, so that FL3'8' and HL3'8' produce identical results.

As we saw in Section 11.8, a *decimal exponent* can be specified in the nominal values in F- and H-type constants. It is written as the letter E followed by an optionally signed decimal integer, as in

```
         DC    F'-43E+6'         –43×10**6, generates X'FD6FDF40'
```

The decimal exponent specifies the power of ten by which the preceding value is multiplied. You could define a table of the first six powers of ten with either of the following two statements:

```
Powers10 DC    F'1,10,100,1000,10000,100000'
Powers10 DC    F'1E0,1E1,1E2,1E3,1E4,1E5'
```

Figure 53. F-type constant with decimal exponent

To improve readability, you can insert blanks among the digits of F-type and H-type constants (remember: *not* in decimal self-defining terms!):

```
Powers10 DC    F'1, 10, 100, 1 000, 10 000, 100 000'
```

In practice, decimal exponents are used mainly in floating-point constants, which we'll discuss in Chapter IX.

You may sometimes want to create *unsigned* or *logical* binary integer constants, as described in Section 2.6 on page 25. You can define such integers by preceding the nominal value of the constant with the letter "U", as in these examples:

```
         DC    F'U2147483648'     X'80000000'  +2**31
         DC    H'U65535'          X'FFFF'      +2**16–1
         DC    F'U4294967295'     X'FFFFFFFF'  +2**32–1
         DC    H'1,U2'            X'00010002'  Mixed signed and unsigned
         DC    H'-1,U32768'       X'FFFF8000'  Mixed signed and unsigned
```

No signs are allowed either before or after the "U".

Exercises

12.1.1.(1)+ In Exercise 11.6.1 on page 142, our friend wanted to define a word binary integer constant with value 10^9. Help him by rewriting the constant with a decimal exponent.

12.1.2.(1)+ Suppose you modified your table of powers of 10 to generate the first six negative powers, as in

```
Powers10 DC    F'1E-0,1E-1,1E-2,1E-3,1E-4,1E-5'
```

What values will be generated?

12.1.3.(2) Suppose you need a halfword constant with value 1/2, so you write

```
Half     DC    H'5E-1'
```

What do you think will be generated?

12.1.4.(1) What will be generated if you write

```
         DC    F'2147483648'      ?
```

12.1.5.(2) What would happen to the Location Counter values in Figure 35 on page 117 if there were now 24 bytes (instead of 22 bytes) between the ST instruction and the first DC instruction?

12.1.6.(2)+ Show the object code generated for these statements:

```
         DC    F'-2147483620'
         DC    H'-32594'
         DC    F'+2147483260'
```

12.2. A-Type Address Constants

The type A *address* constant, sometimes called an "adcon", has great power and broad applicability in Assembler Language programs. An address constant is written differently from the other types we have considered because the nominal value is delimited by parentheses, as in A(10), rather than by apostrophes. Address constants are particularly useful because the nominal value

within the parentheses may be *any expression,* either absolute *or* relocatable.[76] Understanding relocatable address constants involves considering Linker and Program Loader processing, as we will see in Section 39.

A special case where the nominal value of an A-type constant contains a Location Counter Reference is described in Section 13.6 on page 169.

A-type and F-type constants have similarities: a length of four bytes and word boundary alignment are implied. An explicit length suppresses alignment; thus `A(10)` and `F'10'` are equivalent operands, as are `AL4(10)` and `FL4'10'`. The major difference is that you can write *expressions* as the nominal value of constants like `A(X'00012E')` and `A(1+C'.')`. In some contexts, these may be much more natural or convenient than the equivalent F-type constants `F'302'` and `F'76'`.

A-type address constants are especially useful when we want to define word-aligned constants with types not ordinarily aligned by the Assembler. For example, if we need a word containing 1-bits in the rightmost 12 positions and zeros elsewhere, we could write

```
DispMask  DC   A(X'FFF')          X'00000FFF'
```

If we had written this DC's operand field as `XL4'FFF'` instead, we can't guarantee it will be word aligned, even though the same four bytes are generated. Similarly, a word containing the EBCDIC representation of the letter A could be written

```
AConst    DC   A(C'A')            X'000000C1'
```

This is easier to read than `F'193'`, even though the same constant is generated. A constant like

```
Word      DC   A(C'WORD')         X'E6D6D9C4'
```

can be used as a word-aligned “character” constant.

Using such expressions can greatly simplify programming tasks. For example, you can define constants using operands such as

```
Con425    DC   A(ABS425)
```

where the symbol `ABS425` may have been defined in an EQU statement (as in Section 7.6 on page 93) to have a known value. We will see that this technique can provide clarity and simplicity in your programs.

Address constants let you define an area that will contain the *actual* address of a byte in memory when the program is executed. For example, suppose we have written a program that requires the *address* of the word integer constant with value 8, in Figure 51 on page 140. We can define the necessary address constant with the statement

```
Con8Addr  DC   A(Implied)
```

Exercises

12.2.1.(2) Show the generated constant for each of these address constants:

1. `A(X'213'+C'*'-B'11')`
2. `A(92*X'F')`
3. `A(5*C'0'/C' ')`

---

[76] The name “address constant” can be somewhat misleading, because the generated data need not be an address!

12.3. Y-Type Address Constants

Y-type address constants bear the same relationship to A-type adcons as H-type constants bear to F-type constants, except that relocatable Y-type adcons are almost never used. The Y-type adcon has an implied halfword length and alignment, and is identical to the A-type adcon if an explicit length is specified. For example, the operands `H'10'` and `Y(10)` in DC statements define identical 2-byte constants, and the operands `YL1(10)`, `AL1(10)`, `HL1'10'`, and `FL1'10'` all generate `X'0A'`.

If we assume that the symbol **Implied** is relocatable (as in Figure 51 on page 140), then the statement

```
BadCon   DC    Y(Implied)
```

would fail at linking time, because 3 or more bytes will be required to hold the execution-time address of **Implied**.

The main use of Y-type constants is for symbolically-defined constants such as

```
         DC    Y(ABS425)
```

or

```
         DC    Y(C'A')
```

where the equivalent of a halfword integer is desired. Y-type constants are most often used this way: to create a halfword value depending on an absolute expression.

Other address constant types are V, S, and Q. V-type constants are very similar to A-type constants, and will be treated when we discuss external subroutines in Chapter X. Q-type constants will be described when we examine external data structures. The S-type constant generates an addressing halfword that need not be part of an instruction: the value of the operand expression is resolved into base-displacement form. We'll defer these types to later sections.

Exercises

12.3.1.(1) What hex data will be generated by these constants?

```
DC    Y(C'A')
DC    Y(X'F'*C'B')
DC    Y(B'101'*729/C'&&')
```

12.3.2.(3) An S-type address constant is occasionally useful. It has a length of two bytes, which may be implied or explicit. It is almost always aligned on a halfword boundary. The unusual property of this constant is that

```
S(expression)   or   S(expression(expression))
```

is resolved into an *addressing halfword.* For the first (implied address) format, sufficient USING information must be available to the Assembler so that it can resolve the expression into base-displacement form.

Assuming that `A` is a relocatable symbol and that `N` is an absolute symbol, determine the validity of each of the following constants:

`(1) S(A+N), (2) S(A(N)), (3) S(N(7)), (4) S(7(N)), (5) S(N).`

For which of these constants will the result depend (a) on USING information, and (b) on the values of the symbols?

12.4. Constants of Types C, X, and B

Constant types C, X, and B differ in an important way from types F, H, A, and Y: *no defaults are assumed for either length or alignment.* For example, the five bytes required to store the constant generated by the statement

```
         DC    C'12345'
```

will be placed by the Assembler at the next available location given by the current value of the LC. If a particular boundary alignment is desired, we use a DC or DS statement with zero duplication factor, as we'll see in Section 13.2 on page 160.

We write these three constant types almost the same way we write character, hexadecimal, and binary self-defining terms, but the limits on length and value are different. Self-defining terms are restricted to the range between -2^{31} and $+2^{31}-1$ while much longer constants can be defined with the DC instruction.[77] For example, you can define constants as shown in Figure 54.

```
CharCon  DC    C'This is a long character constant'
Digits   DC    X'8462AFCB975310'
ManyBits DC    B'0010111011100011001111011010001011101001'
```

Figure 54. Character, hexadecimal, and binary constants

Note that blanks can be used to separate groups of digits in hexadecimal and binary constants (but not in self-defining terms!) to improve readability. Thus we could write

```
Digits   DC    X'84 62AF CB97 5310'
ManyBits DC    B'0010 1110 1110 0011 0011 1101 1010 0010 1110 1001'
```

The data generated for character (type C) constants is converted to 8-bit bytes using the EBCDIC representation shown in Table 13 on page 87. Blank characters are part of the nominal value, of course! The special rules concerning the apostrophe and ampersand in character self-defining terms also apply to character constants: for each ampersand or apostrophe to appear in the generated constant, a pair of ampersands or apostrophes must appear in the nominal value between the delimiting apostrophes. For example:

```
         DC    C''''              Generates X'7D'
         DC    C'&&'              Generates X'50'
         DC    C'&&&&&&'''        Generates X'5050507D'
```

In Section 7 we noted that the value of a character *self-defining term* is determined by right-adjusting the term in a 32-bit binary field. However, a character *constant* is generated by starting at the *left* end of the character string, and encoding the necessary characters byte by byte. We sometimes say that each byte of a C-type constant contains a character, but it is more precise to say that it contains the 8-bit encoding used to represent the character internally.[78]

Unlike F- and H-type constants, the implied length of C-, X-, or B-type constants is not a fixed number. Because no length modifier is present, the two constants

```
Star1    DC    C'*'          Implied length = 1
     and
Star2    DC    C'**'         Implied length = 2
```

have implied lengths as shown. The Assembler determines the *minimum* number of bytes needed to hold the nominal value of the constant, and assigns that as the implied length of a symbol naming the constant.

This rule also applies to continued constants. For example, in

77 Remember: decimal self-defining terms are always nonnegative!

78 Character representations have many encodings: some are 8, 16, or 32 bits long, and others vary between 1 and 4 bytes! We'll meet some of them in Section 26.

```
ManyChar DC    C'An example of a very long string of characters intende*
               d to illustrate the length attribute of a constant that *
               extends over many lines.'
```

the symbol **ManyChar** has length attribute 134; you certainly don't want to count the characters in each line manually (and possibly make a mistake). It's much easier to use the Assembler's Length Attribute, as in **L'ManyChar**, and know it's correct.

If we write a statement like

```
CharData DC   0C'Characters'
```

the zero duplication factor means that no data will be generated. (We'll discuss this in Section 13.2.) However, the symbol **CharData** will have Length Attribute 10, the length of the nominal value. This method of assigning a Length Attribute to a symbol without necessarily reserving space is often useful.

We will see in Section 27 that the Assembler can generate character constants in other representations such as ASCII and Unicode.

Exercises

12.4.1.(1)+ What are the implied lengths of the constants in Figure 54 on page 150?

12.4.2.(2) How many input lines would be needed to write an Assembler Language statement that defines a B-type constant with an *implied* length of 100 bytes?

12.4.3.(1) How can you specify multiple values in a single operand of a C-type constant?

12.4.4.(2) A four-byte area of memory contains the digit pattern X'4040405C'. What is represented by that pattern? (You should be able to describe two different possibilities.)

12.4.5.(2) Suppose you define the constant

```
         DC    4C' '
```

What is its value if these 4 bytes are thought to represent a binary integer?

12.4.6.(1)+ What is generated for these constants?

```
(1)      DC    B'11110001'
(2)      DC    B'000011111'
(3)      DC    X'0123456'
```

12.4.7.(2) What constants are generated from these statements:

```
 1. DC      C'A''B&&C'
 2. DC      C'''A&&B''''C'
 3. DC      C'ABCF'''
```

12.4.8.(1)+ A programmer wanted to generate 16 bytes of EBCDIC characters representing the 16 possible values of a hexadecimal digit, and wrote

```
EBCHex   DC    X'F0',X'F1',X'F2',X'F3',X'F4',X'F5',X'F6',X'F7',X'F8',X'*
               F9',X'C1',X'C2',X'C3',X'C4',X'C5',X'C6'
```

Can you save some effort for him, and write this in a simpler way?

12.4.9. What constants are generated by these statements? Explain any differences.

```
A        DC    5X'0'
B        DC    XL5'0'

C        DC    5X'7'
D        DC    XL5'7'

E        DC    5C' '
F        DC    CL5' '

G        DC    5C'*'
H        DC    CL5'*'
```

12.4.10. For each of the following sets of statements, the value of the Location Counter is X'000743' when the Assembler encounters the first statement. Give the value and length attributes of all symbols (but not the generated object code).

```
(1)  A        DC    AL3(A)
     B        DC    A(8)

(2)  C        DC    C'DS C''&&'''
     D        DC    C'D DC C''DC'''
```

12.4.11.(1)+ The constant

```
         DC    CL4'345'
```

generates which of these constants?

1. X'00000345'
2. X'00000159'
3. X'F3F4F540'
4. X'00F3F4F5'

12.5. Padding and Truncation

The Assembler must decide what to do if

1. a constant is too *large* to fully occupy the number of bytes allocated for it (whether an explicit length modifier or the default length is used), so some (possibly significant) bits must be *truncated*, or
2. a constant is too *small*, so the generated value must be *padded* to fit in the allotted space.

Some examples are given in Table 31, with the generated constants. Most of the padded constants could have been fit into smaller fields, if you needed desperately to save a few bytes.

Truncation		**Padding**	
Value too large	*Assembled Value*	*Value not too large*	*Assembled Value*
H'65537'	X'0001' (with error!)	H'2'	X'0002'
FL1'+300'	X'2C' (with error!)	FL3'-6'	X'FFFFFA'
CL3'SMITH'	X'E2D4C9' (C'SMI')	CL3'S'	X'E24040'
XL2'56789'	X'6789'	X'56789'	X'056789'
BL1'100100100'	X'24' (B'00100100')	B'101'	X'05' (B'00000101')
AL2(X'789AB')	X'89AB'	A(X'789AB')	X'000789AB'
YL1(X'124')	X'24'	Y(X'124')	X'0124'

Table 31. Examples of truncated and padded constants

For all of the constants on the left, some part of the value must be *truncated* to make it fit in the allotted space, since there is an implied or explicit length in each case. For all these constant types *except* C, excess information is dropped at the *left* end of the constant, and the rightmost portion is assembled. For character constants, the excess is trimmed off the *right* end, as in the `CL3'SMITH'` example above, generating `C'SMI'`. Truncated F- and H-type constants are considered errors by the Assembler.

For the constants on the right side of Table 31 on page 152, more space is allotted either explicitly or implicitly than is needed to hold the significant bits of the given constants. For types H and F, the assembled value is simply the rightmost part of the two's complement representation in which the sign bit has been extended to the left. In the character constant `CL3'S'`, the single letter "S" has been *padded* on the right with two EBCDIC blanks (with representation `X'40'`) to fill out the constant to the required length of three bytes, generating `C'S••'`.[79]

As mentioned in Section 12.4 on page 150, no default length is assumed for constants of types C, X, and B. In the absence of explicit lengths, the Assembler uses just enough bytes for the constant to ensure that no information is lost, and no more. Thus the lengths of the three constants in Figure 54 on page 150 are 33, 7, and 5 bytes respectively; no information is lost, and no padding was required.[80]

Table 32 summarizes some of the rules for writing operands in DC instructions. A complete set of rules is given in the *High Level Assembler Language Reference*. (We'll discuss V-type address constants in Chapter X.)

Type	**H**	**F**	**Y**	**A**	**V**	**B**	**C**	**X**
Maximum Length	8	8	2	4	4	256	256	256
Implied Length	2	4	2	4	4	*	*	*
Implied Alignment	2	4	2	4	4	none	none	none
Value Specified by	dec	dec	absexpr	expr	symbol	bin	char	hex
Delimiters Used	' '	' '	( )	( )	( )	' '	' '	' '
Truncation, Padding	left	left	left	left	left	left	right	left
Multiple Values	yes	yes	yes	yes	yes	yes	no	yes
Note: * The implied length is the minimum number of bytes required to contain the data.								

Table 32. Truncation/padding rules for some DC operands

Section 12.8 on page 157 shows some *type extensions* that let you write longer constants with stricter default alignments.

Exercises

12.5.1.(2)+ What will the Assembler generate for these two statements? Will the results be different? If so, why?

```
DC    CL2'ABC'
DC    AL2(C'ABC')
```

12.5.2.(2)+ Show what will be assembled for each of the following DC statement operands: (1) `F'1000'`, (2) `H'1000'`, (3) `B'1000'`, (4) `XL1'1000'`, (5) `CL1'1000'`, (6) `AL1(1000)`, (7) `YL3(1000)`. Describe the boundary alignment of each.

12.5.3.(2)+ What will be generated for these constants?

79 Remember that we use the • character to represent a blank space.

80 I trust you completed Exercise 12.4.1 before reading this sentence!

```
(1)      DC    B'011110001'
(2)      DC    B'111100010'
(3)      DC    X'01234'
(4)      DC    XL2'012345'
```

12.5.4.(2) The statement preceding Table 31 on page 152 says that some of the constants can be fit into smaller fields. Which ones cannot?

12.6. Literals

We often define data meant to be used *only* as a constant: it should not be modified during program execution. In the sample program in Figure 35 on page 117, the two quantities in the words named **N** and **ONE** are defined by DC instructions, but we expect the symbol **ONE** to mean that the contents of that word retains the value +1 throughout program execution.[81]

Literals are a simple and convenient way to simultaneously define constants and refer to them. A literal is a special kind of *symbol*: the contents of the storage area named by the literal is defined by the “symbol” itself.

A literal is written as an equal sign (=) followed by characters conforming to the rules for a single operand of a DC instruction. These are examples of literals:

```
=F'1'       =C'LongLiteral'       =BL2'111101'
=H'1'       =CL7'BLANK&&'         =X'765432A'
=A(1)       =F'1,2,3,4'           =AL3(5,X'D7'/C'.')
```

Literals may be used in most places where symbols are permitted, with the following exceptions:

1. The Assembler indicates an error if an instruction obviously tries to store into or otherwise modify a constant defined by a literal: thus,

   ```
   ST    7,=F'1'
   ```

   is invalid, even though it's easy to modify “constants” created by the DC assembler instruction statement without any assembly-time indication. (This error detection at assembly time is what makes literals “more constant” than the “constants” defined by DC statements.)

2. A literal may not be specified in an address constant, so that A(=F'1') is invalid.

3. Multiple *operands* may not be specified, but multiple *values* may; thus

   ```
   LM    1,2,=F'1,2'
   ```

   is valid, but both

   ```
   LM    1,2,=F'1',=F'2'
   L     1,=H'1',=H'2'
   ```

   are not, because a literal must be a *single* operand.

4. The duplication factor may not be zero.

5. The alignment and length of the data described in the literal are implied by the constant type, so that this L instruction,

   ```
   L     2,=X'2B'
   ```

   that copies 4 bytes from memory to GR2, will copy unpredictable data into the rightmost three bytes of GR2 because we can't know precisely where the Assembler will place the literal, and what might be in the three bytes following the single byte X'2B'.

---

[81] You can even write statements like

```
ONE      DC    F'137'
```

but this won't make your program easier to understand; and it's even more misleading if your program stores varying values into the word area named **ONE**.

This statement is entirely equivalent to

```
         L     2,X2B
         - - -
X2B      DC    X'2B'             (Not aligned!)
         - - -                   Three more (mystery) bytes
```

except that the symbol **X2B** is not needed when the literal =X'2B' is used.

6. A reference to a literal is always a relocatable implied address (as defined in Section 9.5 on page 109).
7. A literal may be indexed in RX-type instructions, so that

```
         L     0,=F'1,2,3,4,5'(9)
```

is valid, and is exactly equivalent to

```
         L     0,FiveInts(9)
         - - -
FiveInts DC    F'1,2,3,4,5'
```

If the value of the index in GR9 is 8, the L instruction will put the integer 3 in GR0.

8. You may refer to a portion of a literal, as in

```
         IC    0,=F'1'+3
```

but this is considered a very poor programming practice.

9. In most situations, you can use the Assembler's **L'** Attribute Reference notation in an address constant to refer to the Length Attribute of a literal. (Note that this does *not* violate rule 2 above!)

```
LitLen   DC    A(L'=C'This is a message')   Generates X'00000011'
```

which is equivalent to

```
Message  DC    C'This is a message'    Named character constant
MsgLen   DC    A(L'Message)            Length attribute of 'Message'
```

Figure 55. Length attribute reference to two constants, one a literal

The "message" character string is 17 bytes long, but we rarely refer to the Length Attribute of a literal.

We'll make frequent use of symbol length attribute references in Chapters VII and VIII.

After reading this apparently long list of restrictions, you might think that literals are fairly useless. We will see that they can be extremely helpful in writing clear and readable programs, and that these restrictions make good sense.

To illustrate a typical use of a literal, you could rewrite the program segment in Figure 35 on page 117:

```
         BASR  6,0
         USING BEGIN,6
BEGIN    L     2,N
         A     2,=F'1'
         ST    2,N
         ---------
N        DC    F'8'
```

Here, you didn't need to define a constant and create a symbol **ONE** to refer to it.

As literals are encountered in scanning the source program, the Assembler forms a separate internal table containing the literals, with duplicates eliminated. Eliminating duplicates saves space and lets you use literals without generating duplicate constants. The constants from the Assembler's literal table are placed into the program at an appropriate location, and the Assembler then

assigns addressing halfwords to instructions that reference the literals, just as it does for references to symbols.

The area of the program where the Assembler deposits its collection of literal constants into your program is sometimes called a *literal pool.*

Though the Assembler eliminates duplicate literals, those containing references to the Location Counter, as in

```
         L     2,=A(*)
         L     3,=A(*)
```

are not eliminated, because Location Counter values may vary for each occurrence.

For the added ease of referring to constants using literals there is a corresponding loss in your ability to specify exactly where the constant is located, since this is normally determined by the Assembler. The LTORG instruction gives you some control.

Exercises

12.6.1.(2) What data is generated by the literal =AL3(5,X'D7'/C'.') ?

12.6.2.(2) What data is generated by the literal =CL7'BLANK&&' ?

12.6.3.(1)+ Write and assemble a short program containing the statements

```
         DC    2A(*)
T        DC    2A(*-T)
```

and examine the generated object code; describe the differences.

12.6.4.(2)+ In Figure 55 on page 155, the constant named Message is followed by the word-aligned A-type constant named MsgLen. How many bytes might be skipped before the A-type constant?

12.7. The LTORG Assembler Instruction

The LTORG assembler instruction statement lets you control the placement of constants generated by literals. It may have a name-field entry, but no operand field entry. The Assembler aligns the LC at the next doubleword boundary,[82] defines the name-field symbol (if any), and then places its collection of literal-defined constants into the program. The order in which they appear is determined by the Assembler; don't make any assumptions about ordering.

After dumping the contents of its literal table, the Assembler *clears* the table. Excessive use of LTORG instructions in a program with many literals might cause duplicate constants to be defined. For example,

```
         L     0,=F'1'
         LTORG
         L     1,=F'1'
         LTORG
```

will cause two identical constants to be generated.

The literals in the literal pool are generated in decreasing order of alignment. Thus, a word literal like =F'4' will be generated ahead of a halfword literal like =H'2'. This rule applies not only to literals with implied alignment, but to literals whose *length* is a power of two. Thus, the literal =X'00000004' will be generated in the same group as =F'4'.

[82] Or quadword boundary, if the value of the SECTALGN option specifies an alignment stricter than doubleword. See the *High Level Assembler Programmer's Guide* for further information.

This alignment difference can sometimes be surprising. These two constants, though identical, will be aligned differently:

```
         L     2,=FL4'4'          Explicit length 4, word aligned
         L     2,FL4Const         Explicit length 4, not word aligned
         - - -
         DS    F,X                Align LC off a word boundary
FL4Const DC    FL4'4'             Unaligned constant X'00000004'
```

Though rarely a problem, it's worth remembering the difference.

In the absence of any LTORG instructions, the Assembler will generate any accumulated literals at the end of the assembly, so you will need to ensure they are addressable.

We will use literals in many places.

Remember:

A literal is treated by the Assembler as a special *symbol* with the additional effect of causing it to reserve a storage area containing the specified constant.

While the Assembler tries to diagnose instructions appearing to modify a literal, it's easy for your program to modify them by writing into the area where they're stored. (In fact, a program can modify almost anything that's not memory-protected!) You should think of literals as "intended" constants, not as immutable values.

12.8. Type Extensions

As the System z processors have evolved since System/360 was introduced in the mid-1960s, many enhancements and additions have been made to the instruction set and the data types they use.

An important enhancement with z/Architecture was the expansion of the 32-bit general registers to 64 bits, as illustrated in Figure 9 on page 45. To support 64-bit data types, the Assembler extended several existing data types to provide 64-bit constants. Among these are the F-, A-, V-, and Q-type constants. This is done by adding a *type extension* letter following the constant type.

With a "D" type extension, these constants may be up to 8 bytes long, and by default are aligned on doubleword boundaries. For example:

```
         DC    FD'-1'             X'FFFFFFFFFFFFFFFF'
         DC    AD(C'ABC')         X'0000000000C1C2C3'
         DC    FD'U1E15'          X'00038D7EA4C68000'
```

We will see many examples of these doubleword constants when we describe instructions using the 64-bit general registers.

Other type extensions are used for character constants. Many other character representations are now widely used, including ASCII and Unicode. Like EBCDIC, ASCII characters (defined with type extension "A") are one byte long, while the Unicode characters (with type extension "U") generated by HLASM are two bytes long. For example:

```
         DC    C'ABC'             X'C1C2C3'          EBCDIC by default
         DC    CE'ABC'            X'C1C2C3'          EBCDIC always
         DC    CA'ABC'            X'414243'          ASCII  always
         DC    CU'ABC'            X'004100420043'    Unicode
```

The “E” type extension means that the generated constant must use the EBCDIC representation even if the Assembler's TRANSLATE option[83] requests translation of C-type constants to a different encoding. We'll see more about specialized character sets in Section 27.

Other type extensions are used for floating-point data; we'll learn more about them in Chapter IX.

Table 33 summarizes some rules for writing operands in DC instructions with operand type extensions. A complete set of rules is given in the *High Level Assembler Language Reference*.

Type	**FD**	**AD**	**VD**	**QD**	**CA**	**CE**	**CU**
Maximum length	8	8	8	8	256	256	256
Implied length	8	8	8	8	*	*	*
Implied alignment	8	8	8	8	none	none	none
Value specified by	dec	expr	symbol	symbol	char	char	char
Delimiters used	' '	( )	( )	( )	' '	' '	' '
Truncation, padding	left	left	left	left	right	right	right
Multiple Values	yes	yes	yes	yes	no	no	no
Note: * The implied length is the minimum number of bytes required to contain the data.							

Table 33. Truncation and padding rules for some DC operands with extended types

Terms and Definitions

address constant
A field into which a value is inserted by the Assembler, the Linker, or the Program Loader. Typically, an address.

adcon
Abbreviation for “address constant”.

literal
A special symbol with the side effect of defining a constant referenced by that symbol.

literal pool
A set of literal-generated constants grouped together by the Assembler. A program may contain multiple literal pools.

padding
Adding extra bits or bytes to a constant so that it will fill the space allotted to it.

truncation
Removing bits or bytes from a constant so that it will fit in the space allotted to it.

type extension
A second letter following the constant type, providing additional information about the constant's length or representation.

[83] See the *High Level Assembler Programmer's Guide* for details.

13. Data Storage Definition

```
        11      3333333333
       111     333333333333
      1111     33        33
        11               33
        11               33
        11             3333
        11             3333
        11               33
        11               33
        11     33        33
1111111111     333333333333
1111111111      3333333333
```

In this section we examine methods for defining data areas and data structures that simplify programs manipulating the data, and describe the useful assembler instruction statements DS, EQU, and ORG.

13.1. Storage Areas: The DS Assembler Instruction

A storage area is often needed in a program that need not be initialized to contain a value, as done by the DC instruction. This can be done with the DS (“Define Storage”) assembler instruction; it is almost identical to the DC instruction, except that no data is generated: space in the program is allocated, but not initialized. The rules for writing the operand field entry are the same for DC and DS, except that a nominal value (and its enclosing delimiters) is *optional* for DS. Thus the statements

```
         DS    F                  Define word storage
```

and

```
         DS    F'8'               Define word storage
```

both cause the Assembler to reserve a four-byte area on a word boundary, but *no* constant is assembled, even though a nominal value is specified in the second statement. Specifying a value in a DS statement is useful in statements such as

```
         DS    C'Message'         Define storage for characters
```

because it will reserve an area whose length is determined from the length of the nominal value (7 bytes, in this case). Large blocks of storage may be reserved:

```
FW100    DS    100F               Define storage for 100 words
```

This reserves 100 words and assigns the symbol **FW100** to the location of the first. The statements

```
AREA1    DS    80C
AREA2    DS    CL80
```

both define storage areas 80 bytes long, but the Length Attributes of the symbols **AREA1** and **AREA2** are 1 and 80 respectively, which may be very useful in a program. The length attribute of the symbol **AREA1** is 1 byte; the length of the area it names is the product of the duplication factor (80) and the length attribute (1).

In the absence of either a constant or an explicit length for types B, C, and X,

```
     DS  B       and       DS  C       and       DS  X
```

each assigns an implied length of *one* byte and reserves a single byte.

Exercises

13.1.1.(2) Suppose the value of the Location Counter is X'012345' when the following three statements are read by the Assembler:

```
X         DS    AL(4)
Y         DS    A(4)
Z         DS    AL4
```

What is generated by these statements? What are the value and Length Attributes of the symbols X, Y, and Z?

13.2. Zero Duplication Factor

A zero duplication factor may be specified for operands of DS and DC instructions. First, boundary alignment implied by the storage type is performed if necessary. If a name field symbol is present, the aligned value of the LC is assigned as its value; the symbol's Length Attribute is determined from the operand. *No* space is reserved. Thus a DS or DC instruction with a zero duplication factor can be used to force boundary alignment.

For example, the two sets of statements

```
WORD     DS    0F
         DC    C'WORD'
                            and
         DS    0F
WORD     DC    CL4'WORD'
```

both serve to define a four-byte character constant on a word boundary named by the symbol **WORD** which would not in general have been the case if

```
WORD     DC    C'WORD'
                            or
WORD     DC    CL4'WORD'
```

had been specified.

If a zero duplication factor is used in a DC instruction, it behaves just as would the corresponding DS instruction. However, when bytes are skipped to perform alignments required by DS statements, the Assembler does not put zeros into the skipped bytes, while skipped bytes *are* zeroed when aligning DC instructions if the preceding statement generated instructions or data.

Because constants with zero duplication factors do not advance the Location Counter (except for possible boundary alignment), they have many uses. For example, suppose we must define a storage area to hold a (U.S.) ten-digit telephone number:

```
PhoneNum DS    0CL10              Define 10-byte area for full number
AreaCode DS    CL3                Space for area code
Prefix   DS    CL3                Space for prefix
Local_No DS    CL4                Space for local number
```

Figure 56. Describing fields of a (U.S.) telephone number

This way we can refer to the entire field using the symbol **PhoneNum**, or to each component by its name.

Suppose we are writing a program that scans Assembler Language statements, and we want to give names to the fields of the statement. We'll assume that

1. name-field symbols begin in column 1,

2. mnemonics start in column 10 and are 5 or fewer characters long,[84]
3. operands start in column 16 and are less than 20 characters long,
4. remarks lie within columns 36 and 71,
5. column 72 is the continuation column, and
6. columns 73-80 contain sequencing data.

Then, we can name each of the fields of an 80-byte area named **Statemnt** that contains the statement *and* assign appropriate Length Attributes, as shown in Figure 57.

```
Statemnt DS    0CL80              Define 80-column record area
Name     DS    0CL8               Define name-field symbol
         DS    CL9                Reserve space for name-field symbol + blank
Mnemonic DS    0CL5               Define 5-character mnemonic field
Mnemopnd DS    0CL25              Define both mnemonic and operands
         DS    CL6                Reserve space for mnemonic + blank
Operand  DS    0CL19              Define 19-character operand field
         DS    CL20               Reserve space for operand field + blank
Comment  DS    CL36               Allocate 36 columns for comments
Continue DS    C                  Define continuation-indicator column
Sequence DS    CL8                Define sequencing columns
```

Figure 57. Describing fields of an Assembler Language statement

- The first DS statement defines **Statemnt** to be 80 characters long, but reserves no space.
- Similarly, the second DS defines an 8-byte **Name** field beginning at the same location.
- The third DS then causes the Location Counter to be incremented by 9 bytes, so that the symbol **Mnemonic** has a value corresponding to “column 10” of the record.
- Because we might refer to the mnemonic and the operands together, the symbol **Mnemopnd** has the same location, but its length of 25 bytes includes both fields.

The rest of the definitions are similar.

We make this (apparently additional) effort because a program containing these declarations can now refer to the desired fields by *name*. For example, we can use the symbol **Operand** instead of the expression Statemnt+15 to refer to the start of the operand field. While this may not seem an important difference, consider what modifications would have to be made to the program if the **Mnemonic** field is changed to be six characters long: every statement in the program containing a reference to expressions like Statemnt+15 would have to be found and changed.

By using this technique, only the DS statements need changing before the program is reassembled; the statements referring to the various fields in our Assembler Language “statement” need not be changed. Another big advantage of this style of definition is that the symbols have useful Length Attributes; we will see in Chapters VII and VIII how instructions can make good use of that information.

As another example, suppose we wish to reserve space for three words that are also regarded as a single group of twelve bytes named **FWGroup**. We can do this with these statements:

```
         DS    0F                 Align to word boundary
FWGroup  DS    0XL(3*4)           Define start of 3-word group
         DS    3F                 Reserve space for the three words
```

Figure 58. Define a group of words

[84] But: many newer mnemonics can be as many as 8 (or more!) characters long, so you may want to adjust your column positions appropriately. Similarly, the names of some macro instructions we use (like PRINTOUT) are at most 8 characters long.

Exercises

13.2.1.(2) Assemble the statement in Figure 57 on page 161 to verify the locations and lengths of each field. (Remember to add an END statement.)

13.2.2.(3)+ Assume the Assembler's Location Counter is X'345' when it reads each of the following sets of statements. For each symbol, give its value and Length Attribute.

- ```
  J        DS    3H
  K        DS    1X
  L        DS    0F
  ```
- ```
  P        DC    C'A''B'
  Q        DC    2C'ABA'
  R        DC    2A(C'.')
  ```
- ```
  T        DS    XL2'234567'
  V        DC    4Y(37)
  W        DC    0F'1,2'
  ```

13.2.3.(2)+ For each of the following, assume that the Location Counter value is X'345' when the initial statement is processed by the Assembler. Give the value and Length Attributes of the symbol A.

1. ```
   A        DC    F'2'
   ```
2. ```
            DS    0H
            DC    C'*'
   A        DC    C'Asterisk'
   ```
3. ```
            DC    0F'1'
   A        DC    0XL27'0'
   ```
4. ```
   A        DC    A(A)
   ```
5. ```
            DS    19H
   A        DC    X'12345'
   ```
6. ```
            DC    3CL4'ABCDE'
   A        DC    C'A&&B'
   ```
7. ```
            DS    CL400
   A        DC    F'12,34,56'
   ```

13.3. The EQU Assembler Instruction

Two other assembler instruction statements are often useful in defining and describing data areas, EQU and ORG. When we write

```
symbol   Equu   expression
```

the Assembler assigns the attributes of the expression in the operand field (including value, relocatability, and length) to the symbol in the name field.

The EQU instruction reserves *no* storage and generates *no* data or machine language; it only defines a symbol by assigning it an assembly-time value. EQU is a powerful tool for simplifying and understanding programs.

Suppose a program needs a storage area of 75 words, and a word integer constant whose value gives the number of words reserved. The two statements

```
NItems   DC    F'75'             Number of words
Table    DS    75F               Table of words
```

define the necessary items. However, if we decide to change the table size, *both* statements must be changed before re-assembling the program. If we had written instead

```
TblSiz   Equ  75                 Define table size
NItems   DC   A(TblSiz)          Number of words
Table    DS   (TblSiz)F          Table of words.
```

then *only* the EQU statement would have to be modified before re-assembling. If we also want to refer to the word in the "middle" of the table, we can write

```
MidTbl   Equ  Table+(TblSiz/2)*4
```

where the factor 4 is the length of each table entry. This illustrates using EQU to define a relocatable symbol.

A better programming practice is to use the length attribute of **Table** as in L'Table, instead of 4. Here is why: Suppose we can save space in the program by defining halfword table entries instead of words. If we define the symbol **Table** as

```
Table    DS   (TblSiz)H          Table of halfwords.
```

the position of the new table's middle item will still be determined correctly, because the length attribute of **Table** is now 2 instead of 4.

You cannot use EQU instructions to assign more than one value to a symbol.[85] For example, the second statement in this example is invalid:

```
X        Equ  5                  Define X
         - - -
X        Equ  10                 Invalid duplicate definition
```

Exercises

13.3.1.(1)+ Why is the Length Attribute of the symbol **MidTbl** equal to 4?

13.3.2.(2) A programmer wished to conserve space in his program. He needed both a halfword and a fullword binary constant of value +8. He wrote the statements

```
FW8      DC   F'8'
HW8      Equ  FW8+2
```

and referred to the halfword value with the symbol HW8. Can you think of any circumstances in which this might be unsafe?

13.3.3.(2) Suppose the definition of the symbol **Midtbl** had been written in the following forms:

```
MidTbl   Equ  Table+TblSiz*4/2
MidTbl   Equ  Table+4*TblSiz/2
MidTbl   Equ  Table+TblSiz/2*4
MidTbl   Equ  Table+(TblSiz/2)*4
```

Are these equivalent? Why and why not? Why would you choose one in preference to the others?

13.3.4.(2) Describe the differences among the following statements. (It may help to assemble them!) (See Exercise 11.7.1 also.)

85 Some very early assemblers let you use multiple EQU statements to change the value assigned to a symbol. For System z assemblers, the values of *ordinary* symbols are not changed at assembly time. Symbols whose values may be re-assigned at assembly time are called *variable symbols*; they are used in conditional assembly and macros.

```
A1       Equu  5
```

13.3.5.(2) An EQU statement like

13.3.11.(3)+ In each of the following sets of statements, give the value and Length Attribute of each symbol, assuming that the Location Counter value is X'12345' when the first statement of each set is read by the Assembler.

```
1. A        DS    F
   B        DS    2H
   C        DS    2CL2

2. F        DC    A(F)
   G        DC    3AL3(F,G,H)
   H        DC    Y(*-F,275)

3. P        DC    2C'3&&'
   Q        DC    2A(C'3&&')
   R        DS    3XL3'FEDCBA93'

4. X        DC    0FL5'5,10,20'
   Y        DC    FL3'5,10,20'
   Z        DC    2C'5,10,20'
```

13.3.12.(3) Assuming the same statements as in the previous exercise, show the hex data values assembled for the constants having these names: F, G, H, P, Q, and Y.

13.3.13.(2)+ In each of the following sets of statements, give the value and Length Attribute of each symbol, assuming that the Location Counter value is X'01DBC5' when the first statement of each set is read by the Assembler.

```
1. STR      DS    0CL8
   W        DS    2F
   X        DS    2F

2. P        DS    0F
   Q        DS    0H
   R        DC    4X'40'
   S        EQU   *-P
```

13.3.14.(3)+ For each of the following sequences of statements, assume that the value of the LC is X'125' when the first statement is encountered. For each sequence, give the value and Length Attributes of all symbols, the assembled machine language constants (in hex) and their locations, and the LC value after the last statement in the sequence.

```
1.     A     DC    F'-17'
       B     DC    H'33'

2.     D     DC    FL4'+17'
       C     DC    H'-33'

3.     E     DC    C'ABCDEFGH'
       F     DC    F'1000'

4.     G     DS    2H
       H     DC    A(X'129E')

5.     J     DS    0H
       K     DS    0X
       L     DS    0F
       M     DC    0FL6'15'
       N     DC    F'-1000'

6.     P     DC    3C'A''B'
       Q     DC    2A(C'A''B')

7.     R     DS    100F
       S     DS    10CL80

8.     AB    DC    F'900',HL5'2147483650',H'1'
```

```
9.     BC   DC    3XL2'7',0CL3'ABCD',B'1'
       CD   DC    H'16383',H'-16383'

10.    DE   DS    2F,0D,2CL6
       EF   Equ   *,L'DE

11.    T    DC    X'CAB'
       V    DC    2B'101011100'
       W    DC    (V-T)CL(W-T+2)'CAB'

12.    Y    DS    H
       X    DC    (X-Y)AL(X-Y)(X-Y)

13.    Z    DC    CL2'ZZ'
       ZZ   DC    (ZZ-Z)A(ZZ-Z)
```

13.3.15.(2)+ You are given a number N in the range $0 \leq N \leq 14$ and you must use it to assign a pair of symbols **REven** and **ROdd** to an even-odd pair of 32-bit general registers, respectively. Write EQU statements to assign the symbols.

13.3.16.(3) In Exercise 13.3.14, some expressions may be difficult for an Assembler to resolve. Which do you think they are, and why?

13.3.17.(2)+ Suppose a symbol **A** can take values 0 or 1. Write an EQU statement to define a symbol **E** whose value is 1 if **A** is zero, and 0 if **A** is 1.

13.3.18.(3)+ Syppose a symbol **A** can take any value. Write an EQU statement to define a symbol **E** whose value is 0 if **A** is zero, and 1 if **A** has any other value.

13.4. EQU Instruction Extended Syntax (*)

HLASM supports an *Extended EQU Syntax,* allowing you to specify up to five operands.

$\texttt{symbol Equ expression}_1\texttt{,expression}_2\texttt{,expression}_3\texttt{,expression}_4\texttt{,keyword}$

which we understand to mean

```
symbol   Equ   value,length,type,program-attribute,assembler-attribute
```

We have been using only the first operand, expression_1. The second and third operands let you override default values for the length and type attributes.

length (expression$_2$)
: Assigns a new Length Attribute to `symbol`, overriding the Length Attribute assigned from expression_1.

type (expression$_3$)
: Assigns a type attribute to `symbol`. If no **type** operand is present, the Assembler assigns type `U` ("Unknown")

program-attribute (expression$_4$)
: Assigns a programmer-defined "Program Attribute" to `symbol`.

assembler-attribute (keyword)
: A four-character Assembler-defined keyword providing additional information about the expected behavior of `symbol`.

The *High Level Assembler Language Reference* describes the operands in detail. (The last three operands are used mainly for conditional assembly, so we won't discuss them further here.)

The most common use of the Extended EQU statement is to assign specific Length Attributes to symbols. For example, you could write

```
InRec   DS    XL80
OutRec  Equ   *,L'InRec           Length attribute = 80
```

that defines the location of **OutRec** and its Length Attribute. Note that even though the Length Attribute of the Location Counter Reference * would otherwise default to 1 in an EQU statement.

Exercises

13.4.1.(2)+ Assuming that the symbol Result is at location X'2000', give the value and Length Attributes of each symbol.

```
Result   DS    XL133
Pfx      Equ   Result,24
Prod     Equ   Pfx+L'Pfx,12
Cost     Equ   Prod+L'Prod,8
Desc     Equ   Cost+L'Cost,60
Fill     Equ   Desc+L'Desc,(L'Result-L'Pfx-L'Prod-L'Cost-L'Desc)
LFill    Equ   L'Fill
```

13.5. The ORG Assembler Instruction

The ORG instruction lets you modify the Location Counter. Like EQU, it generates no instructions or data. The statement

```
         ORG   expression
```

sets the LC to the value of the expression in the operand field of the statement. The relocatability attribute of the expression must match that of the LC.

We can use the ORG statement to rewrite the data area described in Figure 57 on page 161, as in Figure 59. Note that none of the DS statements uses a zero duplication factor.

```
Statemnt DS    CL80              Define 80-column record area
         ORG   Statemnt          Reset to start
Name     DS    CL8               Define name-field symbol
         ORG   Name+9            Move to 'column 10'
Mnemonic DS    CL5               Define 5-character mnemonic field
         ORG   Mnemonic          Back up the LC
Mnemopnd DS    CL25              Define both mnemonic and operands
         ORG   Mnemonic+6        Move back to 'column 16'
Operand  DS    CL19              Define 19-character operand field
         ORG   Statemnt+35       Move forward to 'column 36'
Comment  DS    CL36              Allocate 36 columns for comments
Continue DS    C                 Define continuation column
Sequence DS    CL8               Define sequencing columns
```

Figure 59. Describing fields of an Assembler Language statement using ORG instructions

After these statements have been processed, the LC will have the value of the expression Statemnt+80, and we can continue assembling as though the LC had never been adjusted by the ORG statements.

Now, suppose we want to check for possible comment statements by defining **Column1** as a new field, so we add the statements

```
         ORG   Statemnt          Back to 'column 1'
Column1  DS    CL1               To check for asterisks
```

at the end of Figure 59. Any statements following the last statement in the figure would begin assembling at Statemnt+1, which is undoubtedly not what you intended.

To rectify such a mistake, you can do either of two things. First, you could place the statement

```
         ORG   Statemnt+80       Move LC to end of Statement field
```

after all the other statements. A second way is to write

```
         ORG   ,                    Set LC to its highest value
```

The Assembler interprets the missing, or null, operand (indicated by the comma) to mean that the LC should be set to the highest value it has attained so far in the assembly.

This example assumes that Statemnt+80 *is* the highest location at this point in the assembly; if not, other instructions and data might be assembled in the wrong places. This possible error is one reason why the technique shown in Figure 57 on page 161 is generally preferred.

The ORG instruction also supports an extended form:

```
         ORG   expression,boundary,offset
```

The Assembler first sets the LC to the location given by “expression”, then rounds it up to the next higher “boundary” (it must be a power of two between 2 and 4096), and then adds the value of “offset” to determine the final LC setting. For example, suppose the current value of the LC is X'12345'. If we write

```
         ORG   *+4,8,-3
```

the Assembler first adjusts the LC to X'12349', then rounds it up to the next doubleword boundary X'12350', and finally subtracts 3, setting the LC value to X'1234D'.

In practice, ORG statements are used infrequently. Their usual applications are to construct data areas that share storage or “overlay” one another, as in Figure 59 on page 167.[86]

Exercises

13.5.1.(2) The programmer mentioned in Exercise 13.3.2 wanted to be as cautious as possible, and changed his constant definitions to

```
FW8      DC    F'8'
         ORG   FW8+2
HW8      DS    H
```

Is this better than the technique used in Exercise 13.3.2? Why or why not?

13.5.2.(2) A programmer didn't know about using a null operand in an ORG statement to reset the LC to its highest value. In trying to do this, he observed that * represents the value of the LC, and therefore wrote

```
Here     Equu  *
         ORG   AnyWhere            Assemble somewhere elsewhere
         - - -
*        Equ   Here                Set LC back to 'Here'
```

What is wrong with this technique? Solve his problem without using an ORG statement with a null operand.

13.5.3.(3)+ In each of the three following code segments, the symbol **A** has value X'982E'. Determine the value and Length Attributes of the symbol **B**.

```
A        DS    29H
B        Equ   A+L'A

A        DS    7CL5
         ORG   A+10
B        DS    2D
```

[86] In higher-level languages, the overlaying of one data definition on another is sometimes called a “union” or a “redefinition”.

```
A        DC    0CL40'*'
         DS    5CL8,3CL3
B        DS    3F
```

13.5.4.(3) With the same assumptions as in Exercise 13.5.3, determine the value, length, and relocatability attributes of the symbol **B**.

```
A        DC    FL7'8'
         ORG   A+2
         DC    HL7'8'
         ORG
B        DC    HL5'-8'
```

```
A        Equ   *
         ORG   A+4*L'A
         DS    (C'*')CL(C'*')'*'
B        Equ   *
```

13.5.5.(3) Using suitable DC and ORG statements, find a way to cause the Assembler to assign the location of some skipped bytes as the value of a symbol **SKIP3**. For example, three bytes are skipped in

```
         DC    F'1',X'2',F'3'
```

13.5.6.(2) Suppose a programmer wrote the statement

```
         ORG                    Set LC to its highest value
```

without a comma to separate the operation and comment fields. What do you expect will happen? Why?

13.5.7.(3) In the instruction sequences illustrated in Figure 57 on page 161 and Figure 59 on page 167, suppose you placed the statement

```
StmtLen  Equ   *-Statemnt
```

following the last statement (with name-field symbol Sequence). What value is assigned to the symbol StmtLen? What value *should* be assigned to StmtLen?

A bonus question: how could you induce the Assembler to detect the difference between the actual and desired values assigned to StmtLen?

13.6. Parameterization

We have seen how we use EQU statements to define quantities such as table sizes, string lengths, and duplication factors; these quantities are *assembly-time constants*, so they are not part of the data whose values may be changed at execution time. The following examples illustrate this.

1. EQU is often used to set a value for defining several storage areas. For example, if you need to process multiple records having the same length, you could define

```
RecLen   Equ   80                 Define record length
InRec    DS    CL(RecLen)         Space for input record
         - - -
OutRec   DS    CL(RecLen)         Space for output record
         - - -
WorkRec  DS    CL(RecLen)         Space for record work area
```

   Then, if the length of the record areas must be changed, you need to modify only the EQU statement and reassemble the program.

2. Suppose a table of five words is stored starting at FTable, and we need to copy the last word of the table into general register 5. We could do this by writing

```
         L     5,FTable+16        Get last word of FTable
```

but we have mixed the data definition (the fact that the word at FTable+16 *is indeed* the last in the table!) with the processing of the data by the L instruction. The “+16” term is a hidden data description.

This program fragment will be easier to understand and modify if we write something like the following:

```
NWords   Equ   5                  Number of words in Table
         L     5,LastWord         Get last word of Table
         - - -
FTable   DS    (NWords)F          Define name and space for Table
LastWord Equ   FTable+(NWords-1)*4     Define last word
```

We can now change the length of the table by modifying the EQU statement, *without* changing the instructions that reference the data. There was nothing in the expression FTable+16 clearly relating it to the number of words in the table. Indeed, if the number of words in the table is less than five, FTable+16 refers to data beyond the end of the table!

3. In Figure 58 on page 161, we might need to change the number of words in the group named **FGroup**. By defining a symbol **NWords** giving the number of words, we can rewrite the example:

```
NWords   Equ   5                  Five-word group this time
         DS    0F                 Align to word
FGroup   DS    0XL(4*NWords)      Length of group
Words    DS    (NWords)F          Reserve space for Words
```

4. Suppose we want to define a table containing a number of character strings, all of the same length. Suppose also that our program processes these strings, *without* knowing in advance either how many there will be or how long they will be. Let **NST** and **STL** be symbols whose values specify respectively the number of strings and the length of each. Then we can reserve storage space for the data with the statement

```
Strgs    DS    (NST)CL(STL)       NST strings each of length STL
```

Then, if we need the addresses of the first and last strings in the block of data, we can define the constants

```
AFirst   DC    A(Strgs)                Address of first string
ALast    DC    A(Strgs+STL*(NST-1))    Address of last string
```

Similarly, if we need halfword integer constants containing the length of each string and the number of strings, we can define these Y-type address constants:

```
HWStL    DC    Y(STL)             Length of a string
HWNSt    DC    Y(NST)             Number of character strings
```

Having written the program to make all its references to the data counts and lengths through these constants, we can finally assign values to the symbols NST and STL by defining two EQU statements:

```
NST      Equ   219                Number of data strings
STL      Equ   43                 Length of a data string
```

As a final example of symbolically-defined data areas, suppose we have written our own Assembler, and have a routine which prints symbol-table information at the end of an assembly. Each line to be printed contains (1) a single carriage-control character to control vertical printer spacing, (2) a symbol up to 8 characters long, (3) a 4-character field for the symbol's length attribute, (4) a 2-character relocatability attribute field, (5) a 4 or 5-character field for the number of the statement in which the symbol was defined, and (6) the rest of the line contains 4 or 5-character fields giving the numbers of the statements whose operand fields refer to the symbol. The fields are to be separated by spaces. In addition, we are to write the definition of the print line so it will work with printers that accept 121 or 133 characters (both are common print-line lengths).

First, we will define the symbols **LineLen** to give the line length (121 or 133), and **StNoLen** to give the number of characters needed to print a statement number (4 or 5). Then, space is reserved for

the “fixed” parts of the line. Finally, we divide the amount of space remaining in the line by the width needed for each reference entry, to determine the number of entries that will fit.

```
LineLen  EQU   133          Assume 133-character print line
StNoLen  EQU   4            Assume 4-character statement numbers
*
StLine   DS    0CL(LineLen)  Start of line
StCC     DS    C            Carriage control character
StSymb   DS    CL8,C        Symbol and trailing space
StLenAt  DS    CL4,C        Length attribute and a space
StRA     DS    CL2,C        Relocatability attribute and a space
StDefn   DS    CL(StNoLen),C Space for defining statement number
*        Number of entries that will fit on rest of line
NXrefs   EQU   ((StLine+LineLen-*)/(StNoLen+1))
*        Define space for references on rest of line
StRefs   DS    (NXrefs)CL(StNoLen+1)  That's all
```

Figure 60. Describing an Assembler symbol cross-reference listing line

The program that uses this print line definition will probably need a constant containing the number of cross-reference entries in the line, so we should also define a constant like

```
MaxRefs  DS    Y(NXrefs)     Maximum number of references
```

to be used while the line is being formatted by the program.

This symbolic technique is important for several reasons.

1. The dependence of individual instructions and constants on the number and length of the data items is more evident when we examine them.
2. If any change must be made to such EQU-dependent quantities, only *one* statement — the defining EQU statement — needs to be changed, and the Assembler will re-calculate all the other quantities depending on it.
3. Statements using the EQU-defined symbols will appear in the Assembler's symbol cross-reference listing.

Experience shows that

Programs are simpler when the definition of data objects is cleanly separated from the instructions that manipulate those objects.

Exercises

13.6.1.(2) Suppose the example above that defines the group of words named FGroup was written

```
NWords   Equ   5               Five-word group this time
FGroup   DS    (NWords-1)F     Space for group
LastWord DS    F               Reserve space for LastWord
```

Determine if this definition gives the same or different results.

13.7. Constants Depending on the Location Counter

We often define address constants with the name of a data item, particularly when we need to provide that address to another program, as in

```
         DC    A(MyData)          Address of “MyData”
```

While it's rare to need the address of a position in a program, as in

```
         DC    A(*)               This address
```

we often need the *offset* of one part of a program relative to another. For example, sometimes it is useful to define constants whose values depend on one another in some regular way. For example, suppose we need a table of 32 bytes containing the binary values 31, 30, 29, ..., 2, 1, 0. We can define the table with an A-type constant:

```
DownTbl  DC    32AL1(DownTbl+31-*)
```

When the first byte is to be generated, * has the same value as the symbol **DownTbl**, so the expression in the constant has value 31. The Assembler does not generate 32 copies of this constant: when a Location Counter Reference appears in an expression in the nominal value of A-type and Y-type constants, the expression is *re-evaluated* before generating each constant.

As each byte is generated, the value of * increases by 1 because the explicit length modifier specifies length 1. Thus the last (32nd) byte will be at DownTbl+31, and the expression will evaluate to 0 as desired.

As another example, suppose we want to build a table of halfword binary integers containing the squares of the integers from 1 to 40. We can write

```
Sqrs     DC    40Y((*-Sqrs+2)*(*-Sqrs+2)/4)
```

where the division by 4 is needed because halfword constants are being generated, so each Location Counter Reference * increases by 2 for each constant generated.

We will find other uses for LC-dependent constants when we discuss data structures in Chapter XI.

Exercises

13.7.1.(3) The Assembler lets you specify an Exponent Modifier for some types of constant, as described in Section 11.8. It is written with the letter E and the value of the modifier immediately preceding the delimiter before the nominal value of the constant. It specifies a power of ten multiplying the nominal value of the constant.

Suppose you want to generate a table of the first ten powers of 10 (starting at 10^0) and you write the statements

```
POWERS   DS    0F
         DC    10FE((*-POWERS)/4)'1'
```

What do you think will happen? Can the Assembler generate the desired constants? If not, what would have to be done to make the Assembler do what you want?

13.7.2.(3)+ Assume the Assembler's Location Counter is X'12345' when it reads each of the following sets of statements. For each of the five symbols, give its value and Length Attribute, and show the generated constants (omitting any *initial* zero bytes inserted by the Assembler for alignment).

```
• A        DC    C'U&&I'
• B        DC    A(*)
•          DS    0XL7C
  C        DC    H'137'
• D        DC    (B'10')AL(B'10')(B'10')
• E        DC    C'A( )',A(C' ')
• F        Equ   *
           DC    2Y(*-F)
```

13.7.3.(2) Analyze the DC statement named **Sqrs** above to determine how it correctly generates a table of squares. Then, write a short program to assemble the statement.

13.7.4.(3) Suppose you want to create a table of 256 bytes in which each byte contains the number of 1-bits in the number representing the byte's offset from the start of the table. For example, the byte at offset 19 should contain 3, the number of 1-bits in B'10011'. Consider the following:

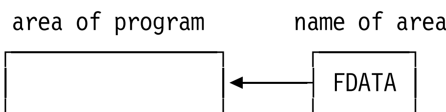
```
T        DC    256AL1(*-T-(*-T)/2-(*-T)/4-(*-T)/8-(*-T)/16-(*-T)/32-(*-X
               T)/64-(*-T)/128)
```

Assemble the statement and verify that it generates the correct values. Then, explain *why* they are correct.

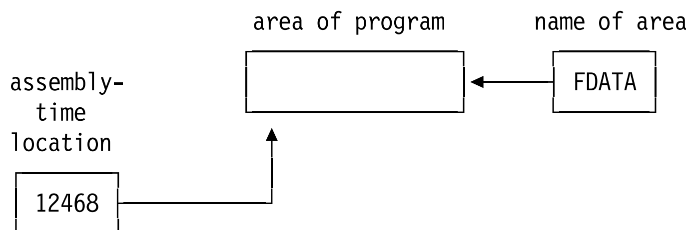
13.8. Assembly Time and Execution Time, Revisited (*)

We will soon investigate many System z instructions that manipulate data, so it is worth reviewing some concepts relating assembly and execution times. Suppose we have an area of a program

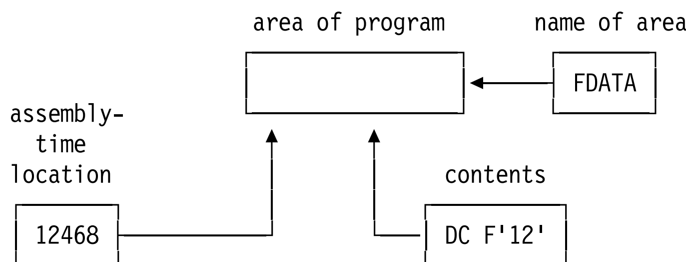
that will eventually contain some data. At assembly time, we give the area a name,



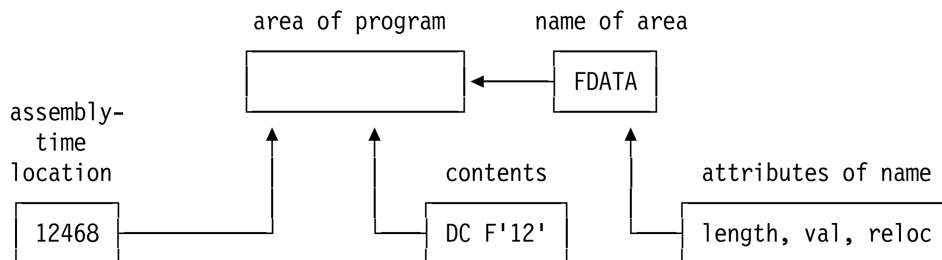
and the Assembler assigns a location.



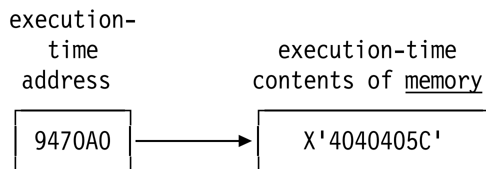
We might also specify a bit pattern for the initial contents of the area.



The *name* of the area has attributes such as value, length, and relocatability, *all of which are distinct* from the value we assign to the bit pattern that is the *contents* of the area.



When the program is executed, this assembly-time information is gone. During loading by the Program Loader, the area of the program is assigned an address in memory. Its contents may have changed as the program is executed.



The “value” of the contents of the area now depends on the context in which it is used. The contents might be treated as instructions, as data of various types, or as commands to be obeyed by an input-output device. The interpretation of the bit pattern depends only on *how* those bits are used, and is not inherent in the bits themselves, nor in any characteristics you assigned to them at assembly time. In this example, the contents of the area of memory may be validly interpreted as (1) an instruction, (2) a word binary integer, (3) a floating-point number, (4) a 9-digit packed decimal number, (5) a character string, and (6) a four-byte bit pattern, among others!

Ideally, the execution-time interpretation of the bit pattern will be the same as the assembly-time interpretation. Instructions will be interpreted only as instructions and not as data, character strings will be used as character strings and not as floating-point numbers, and so on. Assembler Language programming does not always achieve this ideal in practice; but it also gives you *much* more flexibility.[87]

13.9. Summary Observations

As mentioned in the footnote on page 137, the name of the assembler instruction “Define Constant” can be misleading, because the generated machine language data may not be constant during the execution of your program. And even if you intended the data to remain constant, your program may accidentally change its value. (Review the example in Figure 42 on page 122!)

In fact, you can define a “constant” that is actually a machine instruction. For example, if the data generated by the statement

```
         DC    X'1A22'
```

is executed as an instruction, it will add the contents of GR2 to itself.

Another source of confusion may be the fact that you can specify a nominal value in a DS statement, as in

```
NoData   DS    C'This won''t generate any machine language data'
```

There are other contexts where DC statements generate no machine language data, such as Dummy Control Sections (“DSECTs”) and Common (“COM”) sections that we'll discuss in Chapter XI.

There are also potentially misleading names for machine instructions. For example the MVC instruction's name is “Move Characters”, but its operation actually *copies* bytes that may or not be character data.

In summary: don't take the names of assembler and machine instructions too literally. Follow the old advice to “Watch what they do, not what they say”.

Terms and Definitions

EQU Extended Syntax
Additional operands on EQU instructions that provide additional information about the attributes of the symbol defined by the EQU statement.

[87] Some say Assembler Language gives you more “rope you can use to hang yourself”, but that's part of what makes Assembler Language programming fun.

ORG Extended Syntax
Additional operands on ORG statements that allow LC alignment to a specific power-of-two boundary, and an offset from that position.

parameterization
A valuable technique for adding flexibility and generality to program definitions, typically by defining assembly-time constants in EQU statements.

zero duplication factor
A duplication factor that causes LC alignment without generating a constant. Skipped bytes are zeroed for DC instructions if the immediately preceding byte contains object code.

Programming Problems

Problem 13.1.(2)+ Write a program to assemble the DC statements in Section 13.7 on page 171, and verify that the expected constants are generated.

Problem 13.2.(1)+ Write a program in which you define this set of four four-byte binary integers:

```
Ints     DC    F'-1046306171,-1803381883,-1723823710,1082565781'
```

Then, define a 16-byte character string named **Chars** occupying *the same* storage as the four integers. Then, display the 16 bytes of the character string as EBCDIC characters.

Chapter V: Basic Instructions

```
VV        VV
VV        VV
VV        VV
VV        VV
VV        VV
VV        VV
VV        VV
 VV      VV
  VV    VV
   VV  VV
    VVVV
     VV
```

The six sections of this chapter treat instructions basic to almost all Assembler Language programs.

- Section 14 discusses typical instructions that move data between memory and the general registers, and among the general registers. (We'll explore instructions that use data in the Floating-Point Registers in Chapter IX.)
- Section 15 describes the important “Branch on Condition” instructions that let you make decisions about alternate execution paths in your program.
- Section 16 introduces the instructions that perform binary addition, subtraction, and comparison using signed and unsigned binary integer operands.
- Section 17 examines instructions that shift binary numbers in the general registers.
- Section 18 continues our investigation of binary arithmetic operations, examining instructions that multiply and divide numbers in the general registers.
- Section 19 describes instructions performing the logical operations AND, OR, and Exclusive OR on bits in the general registers.

The instructions in this chapter operate on binary data in the general registers, except those in Section 15, which do not involve data.

A comment on terminology: we have used terms like “halfword” and “word” (or “fullword”) to mean data items 2 bytes or 4 bytes long. This has been common usage for many years. However, the *z/Architecture Principles of Operation* uses these terms much more precisely: a halfword must be aligned on a 2-byte boundary, and a word must be aligned on a 4-byte boundary, and similarly for doublewords and quadwords. Please understand that we may use terms like “word” and “halfword” inexactly.[88] Very few instructions require strict alignment of their operands; we will point out those instructions that do require operand alignment.

[88] Correct alignment is always a recommended practice, so our less-than-precise usage isn't usually harmful.

14. General Register Data Transmission

```
   11            44
  111           444
 1111          4444
   11         44 44
   11        44  44
   11       44   44
   11      44444444444
   11     444444444444
   11            44
   11            44
1111111111       44
1111111111       44
```

This section introduces instructions that transmit data among the general registers, and between the registers and memory. We will see instructions that handle data in the 32-bit portion of a 64-bit register, and in the full length of a 64-bit register. (You will remember from Figure 9 on page 45 in Section 3.3 that data items in general registers are frequently manipulated in 32-bit or 64-bit lengths.)

The instructions described here transfer data:

- between the rightmost 32 bits of general registers and memory
- among the rightmost 32 bits of general registers[89]
- between 64-bit general registers and memory
- among the 64-bit general registers.

We'll sample some typical instructions here; there are others we'll see later. The first two groups of instructions leave the high-order half of a 64-bit register unchanged, as illustrated in Figure 61.

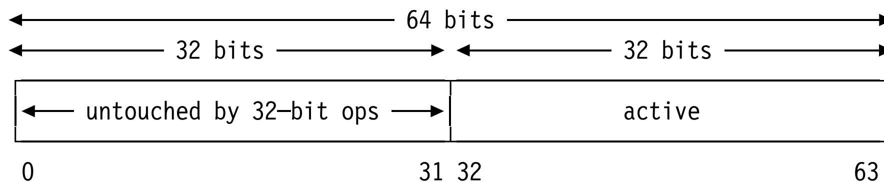


Figure 61. 32-bit portion of a 64-bit general register

The high-order half is “invisible” to the first groups of instructions described here. The System z architects wanted to ensure compatibility with programs that use only 32-bit registers, so that the presence of the high-order bits of the 64-bit register would have no effect on existing programs.

Note: The terms and notations used for various portions of a general register can sometimes be confusing. The *z/Architecture Principles of Operation* uses “High” and “Low” to refer to the

[89] We will occasionally use the older term “32-bit general register” to mean “the rightmost 32 bits of a 64-bit general register.” In System z, all general registers are 64 bits long; before z/Architecture was introduced, general registers were 32 bits long, so the older terminology is still useful for instructions introduced prior to z/Architecture.

left/top/upper and right/bottom/lower portions respectively; but the letters H and L are also used in other contexts with (sometimes) different meanings.

We will use "GR R_1" to mean the general register referenced by the "R_1" operand of a machine instruction statement, and "GRn" to mean "general register n".

The next several sections will describe instructions that affect only the rightmost 32 bits of a 64-bit general register. We'll examine instructions that affect all 64 bits of a register starting in Section 14.7 on page 189.

14.1. Load and Store Instructions

We first examine instructions that transmit data between general registers and memory. The most important are L (Load) and ST (Store), shown in Table 34.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
58	L	RX	Load	50	ST	RX	Store

Table 34. Load/Store instructions for 32-bit general registers

We saw these two instructions in several earlier examples; the operand's Effective Address should be divisible by 4, indicating a word operand.[90] Neither instruction changes the Condition Code.

As a reminder, an RX-type instruction has the form shown in Table 35.

opcode	R_1	X_2	B_2	D_2

Table 35. Format of an RX-type instruction

- The Load instruction L copies 4 bytes of data from memory, starting at the Effective Address, to bits 32-63 of a general register. When executed,

  ```
  L       R1,D2(X2,B2)
  ```

  places a copy of the word at the Effective Address of the assembler instruction statement's second operand from memory into GR R_1. The original contents of GR R_1 are lost, and the word in memory is unchanged. (Remember, "operand" here means both (1) the *assembly* time operand *field* in the assembler instruction statement, and (2) the *data* referenced at *execution* time by the instruction.)

  For example, to set the contents of GR9 to zero, we could write

  ```
  L       9,=F'0'
  ```

  (this is definitely *not* the best way to zero a register, as we will see). To set it to the maximum negative number, we could write

  ```
  L       9,=F'-2147483648'
  ```

- The Store instruction ST copies data from a general register to memory. It is written explicitly as

  ```
  ST      R1,D2(X2,B2)
  ```

  When executed, it causes a copy of the contents of bits 32-63 of GR R_1 to replace the word in memory at the Effective Address of the second operand. The contents of the register are unchanged, and the original contents of the word area are lost.

[90] In the original System/360 systems, correct boundary alignment was required. This requirement was annoying or inconvenient to many programmers, so IBM introduced the "Byte-Oriented Operand Facility" (or "BOOF") to relax the stringent alignment requirement. Correct alignment is still recommended because misaligned operands can sometimes cause programs to run much slower.

For example, to put a copy of the contents of the word at **A** into the word at **B**, we could write

```
L    0,A
ST   0,B
```

and to exchange the contents of the words at A and B, we could write

```
L  1,B         L  0,A         L  0,A            L  0,A
L  0,A    or   L  1,B    or   L  1,B     but    ST 0,B
ST 0,B         ST 0,B         ST 1,A     not    L  0,B
ST 1,A         ST 1,A         ST 0,B            ST 0,A
```

assuming that GR1 is not being used as the program's base register!

L and ST, like other instructions referencing addresses in memory, are subject to interruptions due to addressing and *memory protection*, which provides some control over the areas of memory accessible to a program.

Exercises

14.1.1.(1) What is the difference at assembly and execution times between

```
         L    5,BBB
BBB      EQU  8
```

and

```
         L    5,BBB
BBB      DC   F'8'   ?
```

14.2. Multiple Loads and Stores

We sometimes want to transmit groups of 32-bit words between memory and the right halves of several registers. This can be done with a sequence of L or ST instructions, as in

```
L   1,A                    ST   1,B
L   2,A+4       and        ST   2,B+4
L   3,A+8                  ST   3,B+8
```

If we use more than a very few registers, this is cumbersome and slow. Instead, we use the LM (Load Multiple) and STM (Store Multiple) instructions shown in Table 36. Neither instruction changes the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
98	LM	RS	Load Multiple	90	STM	RS	Store Multiple

Table 36. Multiple load/store instructions for 32-bit general registers

Each is RS-type, for which *three* operands must be specified in the operand field of the assembler instruction statement, as follows:

LM (or STM) $R_1,R_3,D_2(B_2)$ (explicit address)
LM (or STM) R_1,R_3,S_2 (implied address)

The components of the assembled instruction are pictured in Table 37.

opcode	R_1	R_3	B_2	D_2

Table 37. RS-type instruction format

As usual, the assembler instruction statement's R_1 and R_3 operands must be absolute expressions between 0 and 15. The base and displacement may be given explicitly, or derived by the Assembler from an implied address.

Beginning with GR R_1, the CPU stores the contents of registers (for STM) or loads the contents of registers (for LM) in order of increasing register number into or from successive words in memory starting at the Effective Address of the second operand, until GR R_3 has been stored or loaded. If R_3 is less than R_1, then registers GR R_1 through GR15 will be stored/loaded followed by registers GR0 through GR R_3. Thus, register 0 may be considered to "follow after" register 15, so that the general registers "wrap around" from highest to lowest numbered.

Thus, `STM 15,0,X` will store c(GR15) at `X` and c(GR0) at `X+4`, and `LM 15,0,X` will load GR15 from `c(X)` and GR0 from `c(X+4)`.

For example,

```
         LM    2,6,=5F'0'
```

will cause the contents of general registers 2, 3, 4, 5, and 6 to be set to zero. Similarly,

```
         STM   0,15,SAVE
```

will cause the contents of all sixteen registers to be stored beginning at **SAVE**. The symbol **SAVE** could have been defined in a statement such as

```
SAVE     DS    16F
```

This DS instruction ensures correct boundary alignment for the second operand address of the STM instruction. If we assume that GR1 contains the address of a list of four words, we can load them into registers 7 through 10 by executing

```
         LM    7,10,0(1)
```

Similarly, if we assume that register 13 contains the address of a block of 18 contiguous words, then

```
         STM   14,12,12(13)
```

will store registers 14, 15, 0, ..., 12 in successive words, beginning with the fourth word of the given area. While these last two examples may seem contrived, they illustrate parts of common conventions for communicating with subprograms.

As a final example of LM and STM, suppose we wish to exchange the contents of GR0 through GR7, as a group, with the contents of GR8 through GR15. We could write

```
       STM   0,15,SAVE                  STM   8,7,SAVE
       LM    8,7,SAVE       or          LM    0,15,SAVE
       - - -                            - - -
SAVE   DS    16F                 SAVE   DS    16F
```

This ignores one important detail: one of the general registers must have been specified as a base register so that the symbol **SAVE** can be addressed. The STM and LM instructions will work correctly, because the CPU calculates the Effective Address *before* the execute phase of the LM instruction cycle begins. When execution is completed, however, the base register has probably been changed, so either we must inform the Assembler that the base register is changed (with a DROP statement, or a new USING statement), or the correct value must somehow be put back in the original base register.

Exercises

14.2.1.(1)+ Describe the differences between these two instructions:

```
STM   0,0,XXX
ST    0,XXX
```

14.2.2.(2) In describing the STM instruction, we said that

```
STM   14,12,12(13)
```

stores registers 14 through 12 beginning with the *fourth* word of the save area. Explain why it isn't the third, as the displacement value 12=3*4 might imply.

14.2.3.(1) What is the maximum number of general registers whose contents can be modified by a single instruction?

14.2.4.(1) Describe the effect of each of the following instructions:

```
(1)       LM   15,15,X
(2)       STM  0,0,X
(3)       LM   0,0,X
```

14.2.5.(5) Suppose two symbols have been defined with the statements

```
A         EQU   4
B         EQU   9
```

Then, the instruction `STM  A,B,X` stores registers GR4 through GR9 starting at **X**, and we can compute the number of registers stored with the statement

```
NREGS     EQU   B-A+1
```

On the other hand, if we had defined **A** and **B** with

```
A         EQU   9
B         EQU   4
```

then the instruction `STM  A,B,X` would store registers GR9 through GR15 and GR0 through GR4. We can then compute the number of registers stored with the statement

```
NREGS     EQU   B-A+17
```

Thus, the value assigned to **NREGS** depends on what values are assigned to the symbols **A** and **B**. Write an expression in the operand field of the EQU statement that defines **NREGS** such that its value will always tell how many registers were stored by the STM, no matter what values (between 0 and 15) are assigned to the symbols **A** and **B**.

14.3. Halfword Data

Table 38 shows the two instructions described in this section; neither instruction changes the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
48	LH	RX	Load Halfword	40	STH	RX	Store Halfword

Table 38. Halfword load/store instructions for 32-bit general registers

Transmitting halfword data between memory and registers is somewhat more complicated, because a 16-bit halfword requires only half of a 32-bit general register. This may seem obvious, but we need to know (1) which half of the register, and (2) what happens to the other half.

The instructions LH (Load Halfword) and STH (Store Halfword) are similar to L and ST; both are RX-type instructions, and the operand field entry is exactly the same.

STH is simpler: the rightmost 16 bits of GR R_1 replace the halfword in memory at the Effective Address of the second operand, and GR R_1 remains unchanged. If the 32-bit contents of the register is an integer too large to be correctly represented as a 16-bit two's complement integer, the high-order 16 bits are truncated, and significance is lost. *No* indication is made that the halfword in memory may not have the desired value!

When LH transmits data from memory to a general register, the CPU assumes you want to perform arithmetic operations on it, so the result should occupy the entire 32-bit register with the least significant bit at the right-hand end. To give a correct representation in the 32-bit register, *copies of the sign bit* of the 16-bit halfword are *sign-extended* to the left to occupy the left half of the first-operand general register.[91] This is illustrated in Figure 62.

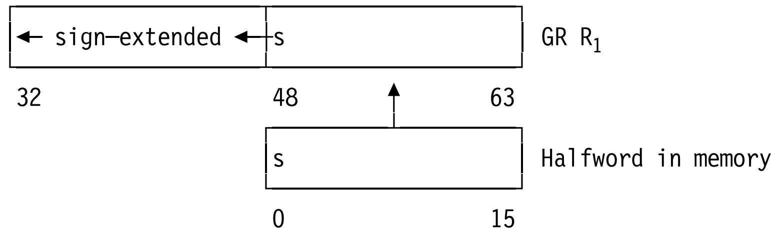


Figure 62. Sign extension by LH instruction

For example, the two statements

```
         LH    0,=H'1'       (=H'1' = X'0001'), c(GR0) = X'00000001'
```

and

```
         LH    0,=H'-1'      (=H'-1' = X'FFFF'), c(GR0) = X'FFFFFFFF'
```

set the contents of GR0 to X'00000001' and X'FFFFFFFF', as indicated. So long as the value of a halfword operand X from memory satisfies

$-2^{15} \le c(X) < 2^{15}$

it can be represented correctly in 16 bits, it will be correctly transmitted by LH and STH instructions. Otherwise, the problems illustrated in the next two examples can occur.

Suppose we execute the instructions in Figure 63. The contents of the registers is given in the remarks fields of the instruction statements.

```
         L    0,=F'65537'         c(GR0)=X'00010001'  +65537 = 2¹⁶+1
         STH  0,A                 c(A)  =    X'0001'  Lost high-order bit!
         LH   1,A                 c(GR1)=X'00000001'  Lost significance!
         - - -
A        DS   H
```

Figure 63. Loss of significant digits using STH/LH

The contents of GR0 and GR1 will be different because the quantity in GR0 stored by the second instruction is too large.

A more awkward result is illustrated in Figure 64.

```
         L     0,=F'65535'        c(GR0)=X'0000FFFF'  +65535
         STH   0,A                c(A)  =    X'FFFF'  No lost bits, but wrong sign
         LH    1,A                c(GR1)=X'FFFFFFFF'  (–1!) Lost significance!
         - - -
A        DS    H
```

Figure 64. Loss of significant digits using STH/LH

In this case, the result in GR1 has sign *and* magnitude different from the original operand.

You can see that when you use halfword data, you must be careful to understand what might happen when storing, loading, or doing (implicitly word) arithmetic with such quantities.

---

[91] We will see many uses of sign extension — copying the sign bit into the higher-order bit positions of a general register — so it's important to understand its behavior.

Exercises

14.3.1.(3) Suppose the STH instruction was modified so that it stored the sign bit and the rightmost 15 bits of the 32-bit R_1 register, so the result contains bits 0 and 17-31 of the original operand. By considering operands like those in Figures 63 and 64, determine whether this form of the instruction will solve some of the problems in using halfword data we've discussed here.

14.3.2.(2) Suppose GR1 contains `X'12345678'`. What will be in GR2 after executing these instructions?

```
ST    1,A
LH    2,A+2
```

Now, suppose GR1 contains `X'FEDCBA98'`; what will be in GR2 after executing the same instructions?

14.3.3.(2)+ Suppose an area of memory contains `X'4040405C'`. Is it an instruction or data? Explain.

14.3.4.(1) What similarities can you find among the opcodes assigned to L, LH, and LM compared to those of ST, STH, and STM?

14.3.5.(3)+ The inequality following Figure 62 on page 183 says that values $\geq 2^{15}$ or $< -2^{15}-1$ can cause problems when used as operands of LH and STH instructions. Write and execute program segments like that in Figure 63 on page 183 to test this assertion.

14.4. Insert and Store Character

The IC (Insert Character) and STC (Store Character) instructions shown in Table 39 transmit a single byte between a general register and memory.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
43	IC	RS	Insert Character	42	STC	RS	Store Character

Table 39. Character insert/store instructions for 32-bit general registers

The operand field entry is written as for L and ST, but you need not worry about boundary alignment for the address of the second operand, since only a single byte of data is being moved.

The instruction

```
STC    R1,D2(X2,B2)
```

stores the rightmost byte of GR R_1 in memory at the Effective Address of the second operand. The contents of GR R_1 and the Condition Code are both unaffected.

The reverse operation, IC, is called "Insert Character" rather than "Load Character", because the byte from memory is *inserted* into the rightmost byte of the register without disturbing the other bytes. No sign extension is done.

Figure 65 on page 185 illustrates the actions of IC and STC.

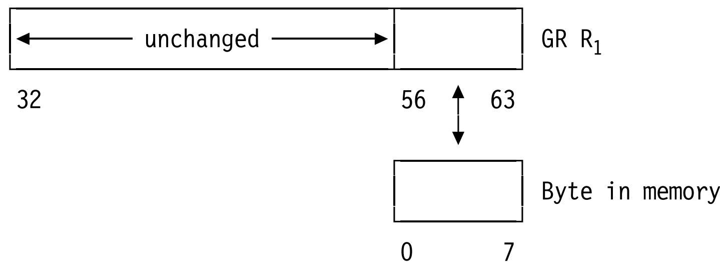


Figure 65. Action of IC and STC instructions

As an example, the instructions in Figure 66 can be used to copy the two characters in the character constant at X, and store them in reverse order at Y.

```
         IC    0,X                Get 1st byte of constant
         STC   0,Y+1              Store at 2nd byte of Y
         IC    0,X+1              Get 2nd byte of constant
         STC   0,Y                Store byte at Y
         - - -
X        DC    C'AB'
Y        DS    CL2                Becomes C'BA'
```

Figure 66. Interchanging two bytes with IC and STC

If memory space is at a premium, you can use a single byte to contain a small integer constant. It can be placed in a register using these instructions:

```
         L     1,=F'0'            Clear GR1
         IC    1,LitlCon          Insert character
         - - -
LitlCon  DC    FL1'53'            Explicit length, no alignment
```

Figure 67. Inserting a small number into a register

but for small constants it is much better to use other available instructions.

Exercises

14.4.1.(3) Write an instruction sequence that will replace the byte at **XX** with a byte that contains in binary the number of one-bits that were present in the original byte. For example, if the initial contents of **XX** was X'48', the final contents should be X'02'. (Hint: define a carefully-constructed 256-byte constant, and use an indexed IC instruction. Show only enough of the constant to clarify how you constructed it.)

14.5. ICM and STCM Instructions

ICM and STCM are very flexible RS-type instructions. They are generalizations of the normal load/store and insert/store character instructions, because you can specify exactly which bytes of a register participate in the “insert” or “store” operation. Table 40 lists the two instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
BF	ICM	RS	Insert Characters Under Mask	BE	STCM	RS	Store Characters Under Mask

Table 40. Insert/Store characters under mask instructions for 32-bit general registers

The final “M” character on these two mnemonics does not mean “Multiple” as in LM and STM, but “Mask” instead.

The instruction format of ICM and STCM is very similar to that of LM and STM, as shown in Table 37 on page 180, but the R_3 digit is now interpreted as a *mask digit* M_3, as illustrated in Table 41. The M_3 operand is a *bit pattern*, not a register number.

opcode	R_1	M_3	B_2	D_2

Table 41. RS-type instruction format for ICM and STCM

The machine instruction statement operand formats for ICM and STCM are like those of LM and STM:

```
         ICM (or STCM)     R1,M3,D2(B2)   (explicit address)
         ICM (or STCM)     R1,M3,S2       (implied address)
```

The four bits in the mask digit correspond to the four bytes of the rightmost 32 bits of the general register designated by GR R_1. The leftmost bit of the mask M_3 (bit 12 of the instruction) corresponds to the leftmost byte of the 32-bit register, the next bit corresponds to the second byte, and so forth. If all mask bits are zero, nothing is inserted or stored.

The CPU executes the STCM instruction by first calculating the Effective Address. Then, where one-bits in the mask appear, the corresponding bytes in GR R_1 are stored into memory in *contiguous* bytes, starting at the Effective Address. Even though separate bytes in GR R_1 may be stored, they are not separated in memory. STCM does not change the Condition Code, and no boundary alignment is required for the second operand.

Suppose the four-byte area of memory named **AA** contains X'01020304', GR12 contains X'FFD0A061', and we execute this STCM instruction:

```
         STCM  12,B'0101',AA      Store bytes 2 and 4 at AA, AA+1
         - - -
AA       DC    X'01020304'
```

The M_3 mask specifies that the second and fourth bytes of GR12 are to be stored into the first two bytes starting at **AA**, so the contents of memory will become X'D0610304'.

STCM can be considered a generalization of the STC, STH, and ST instructions: the three instructions

```
         STC   12,AA              Store rightmost byte at AA
         STH   12,BB              Store 2 rightmost bytes at BB
         ST    12,CC              Store all 4 bytes at CC
```

behave just like these three STCM instructions:

```
         STCM  12,B'0001',AA      Store rightmost byte at AA
         STCM  12,B'0011',BB      Store 2 rightmost bytes at BB
         STCM  12,B'1111',CC      Store all 4 bytes at CC
```

except that now the data areas named by the symbols **BB** and **CC** are not expected to be halfword and word aligned, as recommended for STH and ST. A possible disadvantage of STCM is that it cannot be indexed, since it is not an RX-type instruction.

The ICM instruction performs the inverse operation to STCM, and also does not expect the second operand to be aligned; however, ICM *does* set the Condition Code. As above, suppose the contents of the four bytes in memory in an area named **AA** contain X'01020304', and GR12 initially contains X'FFD0A061'. Then if we execute the instruction

```
         ICM   12,B'0101',AA         Insert into bytes 2 and 4 of GR12
         - - -
AA       DC    X'01020304'
```

the contents of GR12 will become X'FF01A002'.

If all the mask bits are zero, or if all the inserted bytes are zero, the CC is set to zero. Otherwise, the leftmost bit of the *first* byte inserted anywhere into GR R_1 is inspected: if the leftmost bit is a one-bit, the CC is set to 1; otherwise the CC is set to 2. The settings are summarized in Table 42.

CC	Meaning
0	M_3 = 0, or all inserted bytes are zero
1	Leftmost bit of first inserted byte = 1
2	Leftmost bit of first inserted byte = 0

Table 42. CC settings after ICM instruction

This method of setting the CC is easier to understand if we consider the case when the mask digit is all one-bits, meaning that four bytes are brought from memory and placed into GR R_1. If we execute these three instructions, the CC settings are as indicated.

```
ICM    1,15,=F'0'        CC set to 0, c(GR1) is zero
ICM    2,15,=F'-1'       CC set to 1, c(GR2) is negative
ICM    3,15,=F'+1'       CC set to 2, c(GR3) is positive
```

Exercises

14.5.1.(2) Write a sequence of two instructions (using ICM and STCM) that will set the CC to zero if the middle two bytes of GR1 are zero. For example, if c(GR1) = X'A2000064', the CC should be set to zero.

14.6. RR-Type Data Transmission Instructions

We now examine RR-type instructions that transmit data among the 32-bit right halves of the general registers; four of them set CC. The instructions are LR (Load Register), LTR (Load and Test Register), LCR (Load Complement Register), LNR (Load Negative Register), and LPR (Load Positive Register). We saw the LR instruction in the machine instruction statement in Figure 30 on page 78; it is the only one of the five that does not set the CC.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
18	LR	RR	Load Register	10	LPR	RR	Load Positive Register
11	LNR	RR	Load Negative Register	12	LTR	RR	Load and Test Register
13	LCR	RR	Load Complement Register				

Table 43. Register/register instructions for 32-bit general registers

The operand field entry of the instructions is written

R_1,R_2

where R_2 need not differ from R_1. For example,

```
LCR   0,0                 Complement c(GR0)
```

forms the two's complement of the contents of GR0 without affecting any other register.

The action of the first five instructions is summarized in Table 44 on page 188, where the arrow means “replaces” and the vertical bars |...| mean “absolute value”. As noted above, only the rightmost 32 bits of the registers are involved.

Mnemonic	Action	CC Values
LR	c(GR R_1) ← c(GR R_2)	Not changed
LTR	c(GR R_1) ← c(GR R_2)	0,1,2
LCR	c(GR R_1) ← −c(GR R_2)	0,1,2,3
LPR	c(GR R_1) ← |c(GR R_2)|	0,2,3
LNR	c(GR R_1) ← −|c(GR R_2)|	0,1

Table 44. Action of five RR-type general register instructions

The CC is set to indicate the status of the result in GR R_1, as shown in Table 45.

CC	Meaning
0	Result is zero
1	Result is negative
2	Result is positive
3	Result has overflowed

Table 45. Condition Code settings

You can see in Table 44 that the actions of LR and LTR are identical except that LTR sets the CC. We often test the contents of a register by writing instructions like

```
         LTR   4,4
```

that has no effect other than setting the CC. We test the CC with the important "Branch on Condition" instructions we'll see in Section 15.

For LCR, LPR, and LNR, the arithmetic operations use 32-bit two's complement representation. Overflow can occur during execution of LCR or LPR only if c(GR R_2) is the maximum negative number -2^{31}. (It may help to review the discussion of overflow in Section 2.8 on page 27.)

If the overflow condition causes a program interruption, the Interruption Code is set to 8, indicating a Fixed-Point Overflow.[92] No overflow can occur executing LNR because all representable positive values have valid two's complement representations of their negative values.

This example illustrates possible uses of these instructions.

```
*        First, initialize GR2 and GR3
         LM    2,3,=F'1,0'        c(GR2)=1,  c(GR3)=0, CC not set
         LR    7,3                c(GR7)=0,  CC not set
         LTR   2,2                c(GR2)=1,  CC=2
         LNR   1,3                c(GR1)=0,  CC=0
         LCR   4,2                c(GR4)=-1, CC=1
         LPR   0,4                c(GR0)=+1, CC=2
         LNR   5,2                c(GR5)=-1, CC=1
```

Figure 68. Examples of some RR-type instructions

We saw in Section 14.5 on page 185 that these three ICM instructions set the Condition Code as indicated in the comment fields:

```
         ICM    1,15,=F'0'        CC set to 0, c(GR1) is zero
         ICM    2,15,=F'-1'       CC set to 1, c(GR2) is negative
         ICM    3,15,=F'+1'       CC set to 2, c(GR3) is positive
```

[92] This condition is called "fixed-point overflow", to distinguish it from floating-point and decimal overflow. It is one of four program interruptions you can allow or disallow by setting bits in the Program Mask sketched in Figure 12 on page 47. We sometimes say that such disallowed interruption conditions are "masked" or "disabled", and when allowed they are "unmasked" or "enabled". We'll see in Section 16.2.1 how the SPM instruction lets you control these four program interruptions.

These CC settings are exactly what we would have obtained if we had written the six instructions

```
L      1,=F'0'          CC unchanged
LTR    1,1              CC set to 0, c(GR1) is zero
L      2,=F'-1'         CC unchanged
LTR    2,2              CC set to 1, c(GR2) is negative
L      3,=F'+1'         CC unchanged
LTR    3,3              CC set to 2, c(GR3) is positive
```

That is, an ICM instruction whose mask is all one-bits is equivalent to a L instruction followed by an LTR instruction,.

Unfortunately, this parallel is invalid for the LH instruction, because ICM does not extend the sign bit to the left to fill the register as does LH. The ICM instruction in

```
L      1,=F'0'          Set GR1 to zero, CC unchanged
ICM    1,B'0011',=H'-1'  Sets GR1 to X'0000FFFF', CC = 1
```

sets the CC to 1 (indicating a one-bit at the left end of the first inserted byte), but the leftmost two bytes of GR1 are still zero. Conversely, the instruction

```
LH     1,=H'-1'         Set GR1 to X'FFFFFFFF'
```

does not affect the CC, but GR1 will contain all one-bits.

Exercises

14.6.1.(2) What changes to the Assembler would be needed to let you use a (nonexistent) “STR” opcode (that is, “Store Register”, in the same sense as “Load Register”)?

14.6.2.(2) For each instruction in Table 43 on page 187, what operands in GR R_2 can result in the CC being set to 3?

14.7. Load, Store, and Insert for 64-bit General Registers

We'll now look at instructions that manage data using the full 64 bits of a general register. Contrast Figure 69 below with the 32-bits-only view in Figure 61 on page 178.

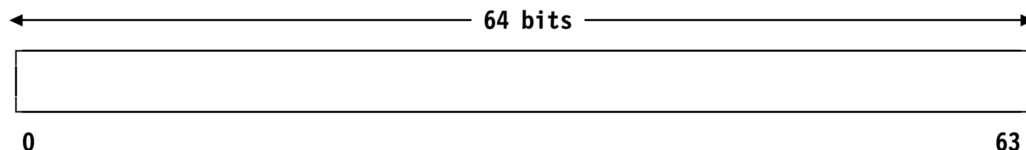


Figure 69. 64-bit general register

The instructions are shown in Table 46:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E304	LG	RXY	Load	E324	STG	RXY	Store
E315	LGH	RXY	Load Halfword (64←16)				
EB04	LMG	RSY	Load Multiple	EB24	STMG	RSY	Store Multiple
EB96	LMH	RSY	Load Multiple High	EB26	STMH	RSY	Store Multiple High
EB80	ICMH	RSY	Insert Characters Under Mask (High)	EB2C	STCMH	RSY	Store Characters Under Mask (High)

Table 46. Register/storage instructions for 64-bit general registers

The letter "G" is used in almost all the instructions involving 64-bit registers. For example, LG and STG are the 64-bit equivalents of L and ST.[93]

Here, we introduce two variations on the RX and RS formats. RXY-type instructions behave just like RX-type instructions, except that they provide a longer, and signed, displacement field, as shown in Table 47.

opcode	R_1	X_2	B_2	DL_2	DH_2	opcode

Table 47. RXY-type instruction format

Another instruction type is RSY. Its format and behavior are very similar to RS-type instructions, and it shares the "long-displacement" format with RXY-type instructions.

opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

Table 48. RSY-type instruction format.

For now, we'll treat both RXY-type and RSY-type instructions as though they are identical to RX-type and RS-type instructions, because they do very similar things. Also note that the first four bytes of RX-type and RS-type instructions have the same format as the first four bytes of RXY-type and RSY-type instructions, respectively.

We'll investigate the added usefulness of the "long-displacement" instructions and their "DL_2" and "DH_2" fields (and how the Assembler handles them) in Section 20.1 on page 302.

We've seen instructions that manipulate data only in the low-order 32 bits of a general register, while others deal with all 64 bits. To help us distinguish these two views of a general register, we introduce another notation, GR G_n. Thus, GG R_1 will mean the *64-bit* general register referenced by an R_1 operand, while GR R_1 will continue to mean the *32-bit* general register referenced by an R_1 operand. Similarly, GGn will mean the specific 64-bit register referenced by GR G_n, and GRn will mean the specific 32-bit register referenced by GR R_n.

The LG, STG, LMG, and STMG instructions do for 64-bit registers exactly the same actions as their 32-bit equivalents L, ST, LM, and STM.

1. To illustrate STMG, suppose we save 64-bit general registers GG0 through GG3 at Save0123:

```
          STMG  0,3,Save0123       Save 64-bit GG0 through GG3
          - - -
Save0123  DS    4D                 Reserve 4 doublewords
```

   In memory, these would appear like this:

Save0123+0	c(GG0)
Save0123+8	c(GG1)
Save0123+16	c(GG2)
Save0123+24	c(GG3)

[93] In my opinion, "G" was chosen for two reasons. First, it was the least-used letter among the nearly 500 instruction mnemonics supported by Enterprise System/390 architecture, the predecessor of System z. Second (and anecdotally), the largest size latte sold at a coffee shop near IBM in Poughkeepsie, New York was called a "**G**rande", so it seemed natural to say the new large registers were similarly "Grande". Other more descriptive letters like "L" (meaning "Long") were already used in many other mnemonics, where "L" can mean "Logical", or "Long", or "Low".

Then, to restore the contents of the registers, we execute

```
         LMG   0,3,Save0123        Restore GG0 through GG3
```

2. The other two LM/STM instructions in tref refid=s14s8t1. end in “H”, referring to the 32-bit high-order half of a 64-bit general register.[94] They do the same actions for the high-order halves of 64-bit general registers that LM and STM do for the low-order halves. LMH and STMH might seem unnecessary, since LMG and STMG manage *both* halves of a 64-bit register in one operation. The reason they exist is “history”.[95]

   For example, to store and load only the high-order halves of general registers 5 and 6, we can write

```
         STMH  5,6,High56          Save high-order half of GG5 and GG6
         - - -
         LMH   5,6,High56          Restore high-order half of GG5 and GG6
         - - -
High56   DS    2F                  Save area for two 32-bit words
```

3. LGH is the 64-bit equivalent of LH: it sign-extends the 16-bit integer at the Effective Address in memory to 64 bits, and places the result into GG R_1.

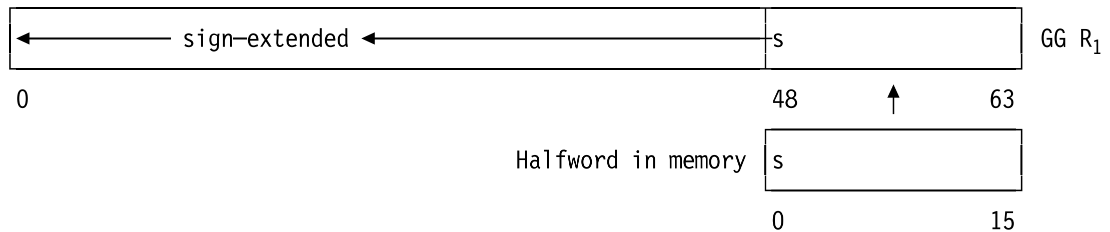
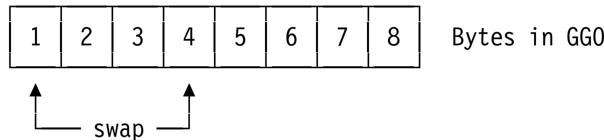


Figure 70. Sign extension by LGH instruction

4. The remaining two instructions in Table 46 on page 189 are ICMH and STCMH. They behave exactly like ICM and STCM, except that the four M_3 mask bits now refer to the four bytes in the *high*-order or left half of GG R_1. The Condition Code settings are the same as in Table 42 on page 187.

   To illustrate, suppose you want to swap the first and fourth bytes of GG0 — that is, the high-order and low-order bytes of the high-order half of the register.



   These three instructions show one way to do this:

```
         STCMH 0,B'1001',Temp      Save bytes 1 and 4 of 64-bit GG0
         ICMH  0,B'1000',Temp+1    Insert original byte 4 at left end
         ICMH  0,B'0001',Temp      Insert original byte 1 into 4th byte
         - - -
Temp     DS    XL2                 Two-byte temporary storage
```

[94] The letter “H” is used in mnemonics with many meanings, such as “High”, “Halfword”, etc.

[95] Because 64-bit general registers were introduced after many years of program development using only 32-bit general registers, conventions for saving and restoring the 32-bit registers are embedded in many programs. To minimize the changes needed, programs can save the low-order register halves using existing conventions, and then save the high-order halves elsewhere using STMH. The LMD instruction, as we will see, lets you restore both halves of 64-bit registers from two separate save areas in one operation.

Exercises

14.7.1.(1)+ There is a LGH instruction, but no STGH instruction. Why not?

14.7.2.(2) Write a sequence of instructions to exchange the high-order and low-order halves of GG0.

14.8. RRE-Type Data Transmission Instructions for 64-bit General Registers

As the original System/360 architecture evolved, there were not enough one-byte RR-type opcodes available for new register-to-register instructions, so new two-byte opcodes were assigned using a new instruction type, RRE. RRE-type instructions are four bytes long (while "traditional" RR-type instructions are two bytes long). The 8-bit "unused" field in Table 49 is set to zero by the Assembler.

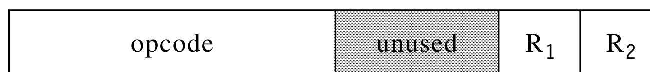


Table 49. RRE-type instruction format

The instructions in Table 50 are RRE-type.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B904	LGR	RRE	Load Register (64)	B900	LPGR	RRE	Load Positive Register (64)
B901	LNGR	RRE	Load Negative Register (64)	B902	LTGR	RRE	Load and Test Register (64)
B903	LCGR	RRE	Load Complement Register (64)				
B927	LHR	RRE	Load Halfword	B907	LGHR	RRE	Load Halfword (64←16)

Table 50. Register/register instructions for 64-bit general registers

The actions of the instructions in Table 51 are the same as for their 32-bit equivalents in Table 44 on page 188.

Mnemonic	Action	CC Values
LGR	c(GG R_1) ← c(GG R_2)	Not changed
LTGR	c(GG R_1) ← c(GG R_2)	0,1,2
LCGR	c(GG R_1) ← −c(GG R_2)	0,1,2,3
LPGR	c(GG R_1) ← |c(GG R_2)|	0,2,3
LNGR	c(GG R_1) ← −|c(GG R_2)|	0,1

Table 51. Action of five RR-type 64-bit general register instructions

The instructions in Figure 68 on page 188 dealt with data in 32-bit registers; their equivalents for 64-bit registers are shown in Figure 71.

```
*        First, initialize GG2 and GG3 (all 64 bits)
         LMG   2,3,=FD'1,0'      c(GG2)=1,  c(GG3)=0, CC not set
         LGR   7,3               c(GG7)=0,  CC not set
         LTGR  2,2               c(GG2)=1,  CC=2
         LNGR  1,3               c(GG1)=0,  CC=0
         LCGR  4,2               c(GG4)=-1, CC=1
         LPGR  0,4               c(GG0)=+1, CC=2
         LNGR  5,2               c(GG5)=-1, CC=1
```

Figure 71. Examples of some RR-type instructions for 64-bit operands

If we compare Figures 71 and 68, we see that these equivalent instructions behave similarly and produce identical CC settings.

LHR is similar to LR, and their operand field entries are the same. LHR takes the rightmost 16 bits of the R_2 register, extends the sign bit to the left to form a 32-bit result, and places it in the R_1 register. This is illustrated in Figure 72; note its similarity to Figure 62 on page 183. The R_1 and R_2 registers need not be distinct.

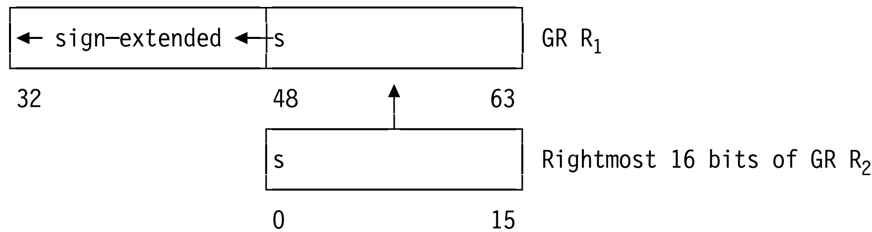


Figure 72. Sign extension by LHR instruction

For example:

```
         L     1,=X'456789AB'
         LHR   2,1                c(GR2)='X'FFFF89AB'
```

We saw in Figure 63 on page 183 that large fullword values can yield incorrect values if truncated to halfwords. The same problem can occur with LHR:

```
         L    0,=F'65537'         c(GR0)=X'00010001'
         LHR  1,0                 c(GR1)=X'00000001'  Lost significance!
```

LGHR uses the rightmost 16 bits of the second operand register. Sign extension is indicated by the notation (64←16) in tref refid=s14s8t1.. Figure 70 on page 191 shows its resemblance to LH seen in Figure 62 on page 183.

Exercises

14.8.1.(2) Can you think of a reason why an LHR instruction would be used with identical register operands?

14.8.2.(2)+ Suppose GR1 contains X'12345678'. What will be in GR2 after executing this instruction?

```
         LHR   2,1
```

Now, suppose GR1 contains X'FEDCBA98'; what will be in GR2 after executing the same instruction?

14.9. The Load and Test Instructions

In Section 14.6 on page 187, we saw two ways to transfer a data item from memory to a general register and set the CC depending on its sign:

```
         ICM   R1,B'1111',dataname
and
         L     R1,dataname
         LTR   R1,reg
```

ICM cannot be indexed nor can it be used for 64-bit operands, because ICM and ICMH set the CC separately for the low-order and high-order halves of GG R_1, respectively. The L/LTR and LG/LTGR instruction pairs can be indexed, but two instructions are needed.

To eliminate these inconveniences, System z provides the LT and LTG instructions, as shown in Table 52 on page 194:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E312	LT	RXY	Load and Test (32)	E302	LTG	RXY	Load and Test (64)

Table 52. Load and Test instructions

Their behavior is identical to the instruction pairs L/LTR and LG/LGTR, respectively.

Exercises

14.9.1.(1) How can the L/LTR and LG/LTGR instruction pairs be "indexed"?

14.9.2.(1) What operand values can cause LT or LTG to set CC=3?

14.10. Mixed 32- and 64-bit Operands

To make it easy to use 32-bit binary operands in 64-bit operations, System z provides a set of instructions that automatically sign-extend a 32-bit operand to 64 bits. In Table 53, The LGF instruction is RXY-type and the others are RRE-type. The notation (64←32) indicates the extension of the 32-bit second operand to a 64-bit first operand.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E314	LGF	RXY	Load (64←32)	B914	LGFR	RRE	Load Register (64←32)
B912	LTGFR	RRE	Load and Test Register (64←32)	B913	LCGFR	RRE	Load Complement Register (64←32)
B910	LPGFR	RRE	Load Positive Register (64←32)	B911	LNGFR	RRE	Load Negative Register (64←32)

Table 53. Register/register instructions for 64-bit general registers

The actions of these instructions are *almost* the same as their 64-bit equivalents that we saw in Table 51 on page 192, except that no instruction sets the CC to 3.

Mnemonic	Action	CC Values
LGF	c(GG R_1) ← c(Word in memory)	Not changed
LGFR	c(GG R_1) ← c(GR R_2)	Not changed
LTGFR	c(GG R_1) ← c(GR R_2)	0,1,2
LCGFR	c(GG R_1) ← −c(GR R_2)	0,1,2
LPGFR	c(GG R_1) ← |c(GR R_2)|	0,2
LNGFR	c(GG R_1) ← −|c(GR R_2)|	0,1

Table 54. Action of 32-bit-to-64-bit general register instructions

In each case, the 32-bit second operand is first sign-extended internally, and then treated as a 64-bit operand. For example, the single instruction

```
         LCGFR 0,1                 Sign extend, complement GR1 to GG0
```

is equivalent to the two instructions

```
         LGFR  0,1                 Sign extend GR1 to GG0
         LCGR  0,0                 Two's complement of GG0
```

(See Exercise 14.10.2!)

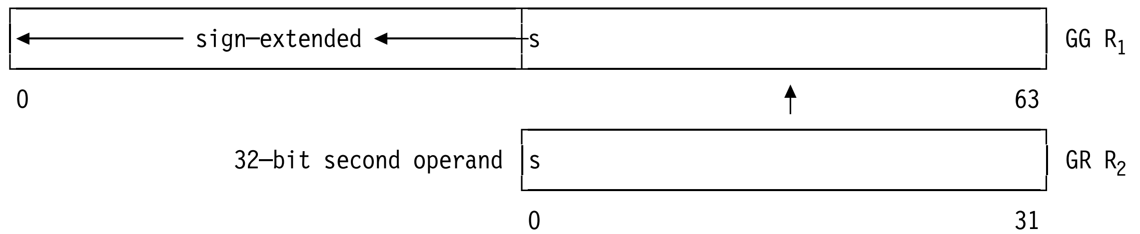


Figure 73. Sign extension for instructions with mixed 32- and 64-bit signed operands

The instructions in Table 53 on page 194 all have the letter “F” in their mnemonics, to indicate that the second operand is a 32-bit, 4-byte “Fullword”. The first operand (designated by R_1) is the 64-bit general register that receives the sign-extended (and possibly complemented) second operand.

Exercises

14.10.1.(1) Compare the opcodes of the five RRE-type instructions in Table 53 on page 194 to those in Table 50 on page 192. What similarities and differences do you see?

14.10.2.(2)+ What would happen in the (64←32) instructions in Table 53 on page 194 if complementation is done before sign extension?

14.10.3.(2)+ The Condition Code values shown in 51 are not the same as those shown in Table 54 on page 194. How and why are they different?

14.10.4.(2) Consider the 32-bit maximum negative number `X'80000000'`. Show the contents of general register 0 and the CC setting after each of these instruction sequences:

```
(1)      L     0,=X'80000000'
         LPR   0,0

(2)      L     0,=X'80000000'
         LGFR  0,0

(3)      L     0,=X'80000000'
         LPGFR 0,0

(4)      L     0,=X'80000000'
         LNGFR 0,0
```

14.10.5.(2)+ Why can none of the instructions in Table 53 on page 194 set the CC to 3?

14.11. Other General Register Load Instructions (*)

The previous sections examined the most frequently used load instructions; System z supports many others. For example, many programs need to insert a byte into the rightmost 8 bits of a general register; a common instruction pattern is

```
         L     1,=F'0'       Clear GR1 to zero
         IC    1,Byte        c(GR1) = X'000000xx'
or
         SR    1,1           Set c(GR1) to zero (subtract from itself)
         IC    1,Byte        c(GR1) = X'000000xx'
```

An extra instruction is needed to set GR1 to zeros before the IC instruction. (The SR instruction subtracts c(GR1) from itself; we'll review it in more detail in Section 16.2.)

The LLC instruction (“Load Logical Character”) does both operations: the byte is loaded into the last 8 bits of the GR1 operand, and the rest of GR1 is cleared to zero:

```
LLC   1,Byte          c(GR1) = X'000000xx'
```

Table 55 gives a summary of the instructions we'll review. We'll see that they are arranged in simple groups with repeating patterns.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E376	LB	RXY	Load Byte (32←8)	B924	LBR	RRE	Load Byte (32←8)
E377	LGB	RXY	Load Byte (64←8)	B906	LGBR	RRE	Load Byte (64←8)
E394	LLC	RXY	Load Logical Character (32←8)	B994	LLCR	RRE	Load Logical Character (32←8)
E390	LLGC	RXY	Load Logical Character (64←8)	B984	LLGCR	RRE	Load Logical Character (64←8)
E395	LLH	RXY	Load Logical Halfword (32←16)	B995	LLHR	RRE	Load Logical Halfword (32←16)
E391	LLGH	RXY	Load Logical Halfword (64←16)	B985	LLGHR	RRE	Load Logical Halfword (64←16)
E317	LLGT	RXY	Load Logical Thirty One Bits (64←31)	B917	LLGTR	RRE	Load Logical Thirty One Bits (64←31)
E316	LLGF	RXY	Load Logical (64←32)	B916	LLGFR	RRE	Load Logical (64←32)

Table 55. Other general register load instructions

For example, the four instructions in the first two rows are *arithmetic* loads: the high-order bit of the second operand is sign-extended to the length of the first operand (as illustrated in Figure 62 on page 183 and Figure 70 on page 191); the others are *logical* load instructions that zero-extend the second operand to the length of the first.

None of the instructions in Table 55 change the Condition Code.

14.11.1. Load Byte Instructions

The LB, LBR, LGB, and LGBR instructions treat the second operand as a *signed* 8-bit number, and sign-extend it to the 32- or 64-bit length of the R_1 general register operand. For the LGB and LGBR instructions, this is illustrated in Figure 74.

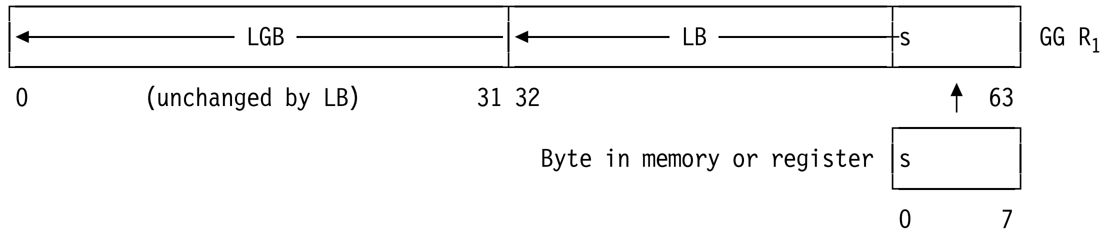


Figure 74. Sign extension by Load Byte instructions

For the LB and LBR instructions, the R_1 first operand is a 32-bit general register, and the high-order 32 bits of the 64-bit register are unchanged. For example:

```
LB    3,=FL1'-7'        c(GR3)=X'FFFFFFF9'
LGB   5,=FL1'-7'        c(GG5)=X'FFFFFFFF FFFFFFF9'
```

14.11.2. Load Logical Character Instructions

The second operand of these instructions is called an unsigned “character” to distinguish it from the (signed) “byte” operand of the Load Byte instructions. Each of LLC, LLCR, LLGC, and LLGCR does the same as the “Byte” instructions above, except that the rest of the R_1 first operand register is set to zeros. To illustrate LLGC and LLGCR:

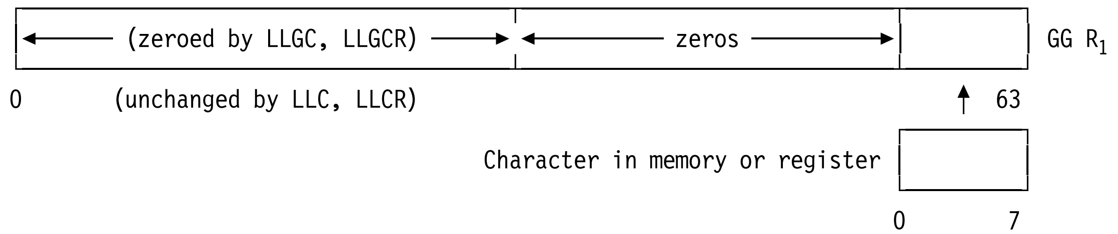


Figure 75. Zero extension by Load Logical Character instructions

LLC and LLCR affect only the 32 low-order bits of the 64-bit R_1 register; the high-order 32-bits are unaffected.

These four instructions can eliminate the need to clear a register before inserting a character for processing.

14.11.3. Load Logical Halfword Instructions

The four instructions LLH, LLHR, LLGH, and LLGHR all load the 16-bit halfword operand into the low-order 16 bits of the R_1 register, and clear the preceding 16 bits of the 32-bit register (for LLH and LLHR) or the preceding 48 bits of the 64-bit register (for LLGH and LLGHR).

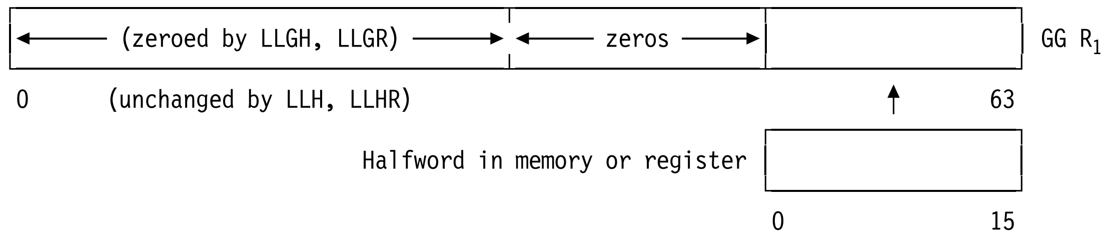


Figure 76. Operation of Load Logical Halfword instructions

These four Load Logical Halfword are closely related to the arithmetic “Load Halfword” instructions, except that the logical loads fill the rest of the 32- or 64-bit R_1 register with zeros, rather than sign-extending the high-order bit of the loaded halfword.

14.11.4. Load Logical (Word) Instructions

The LLGF and LLGFR instructions load the 32-bit second operand from memory or from GR R_2 into the low-order 32 bits of the 64-bit GG R_1 register, as illustrated in Figure 77:

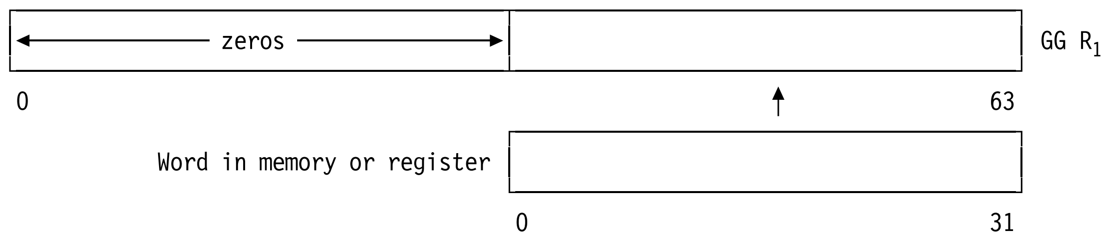


Figure 77. Operation of Load Logical word instructions

In effect, LLGF and LLGFR are like L and LR, followed by setting the high-order half of the GG R_1 register to zero. The R_1 operand is always a 64-bit register.

14.11.5. Load Logical Thirty One Bit Instructions

The LLGT and LLGTR are unusual: the second operand is 32 bits long, but its high-order bit is ignored! This is illustrated in Figure 78 on page 198:

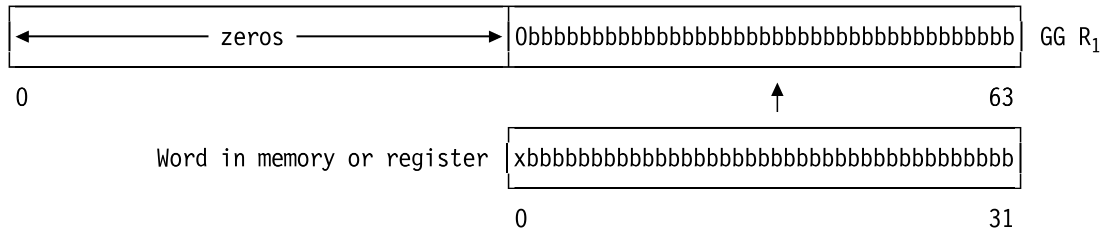


Figure 78. Operation of Load Logical Thirty One Bits instructions

The R_1 operand is always a 64-bit register.

Why ignore the high-order bit of the 32-bit second operand? When we discuss "addressing modes" and instructions that both change and depend on addressing mode in Section 20, we'll see that the high-order bit of a 32-bit address was often used to indicate which mode was desired; it can be important to set that bit to zero.[96]

Exercises

14.11.1.(1) How would you simulate the action of the Load Logical Character instructions with other instructions already described?

14.11.2.(1) How would you simulate the action of the Load Logical (Word) instructions with other instructions already described?

14.12. Misunderstandings to Avoid

Two common errors made by beginning programmers are

1. confusing the LR and L instructions, and
2. trying to use a "Store Register" or "STR" instruction to "store" one register into another.

First: by substituting L for LR, you can occasionally create an error that can't be detected by the Assembler, and sometimes is difficult to find. For example, suppose you intended to load GR5 from GR8 with a LR instruction. If the symbols **R5** and **R8** have values 5 and 8, then both

```
         L     5,8                Load GR5 with value 8 (??)
```
and
```
         L     R5,R8              Load GR5 from GR8 (??)
```

are valid instructions referring to *memory* at address 8. This is probably not what was intended for the second instruction, even though it looks like it is "loading" GR5 from GR8. This can be seen by checking the machine language object code generated for the two instructions:

```
000000 5850 0008          L     5,8              Load GR5 with value 8 (??)
000004 5850 0008          L     R5,R8            Load GR5 from GR8 (??)
```

Exactly the same instruction will be executed.

> **Warning**
>
> Symbols like R8 — an "R" followed by a number — might not refer to a register!

96 This unusual behavior is difficult to justify now, but we'll see in Section 37 that there are good reasons to have these instructions.

To help you remember the difference between related instructions of different types, note that almost all RR-type instruction mnemonics end in the letter "R", while the RX-, SI-, and RS-type instruction mnemonics end in other letters.

Second: there is no "STR" instruction. To "store" data from one general register to another, you must use a LR-like instruction. (See Exercise 14.6.1.)

14.13. Summary

You can think of load-type instruction statements as moving data from right to left: that is, the second operand replaces the first operand. For example, consider this assembler instruction statement:

```
L     0,X                 c(GR0) ←— c(X)    Right to left
```

This way of visualizing actions applies to most instructions. The primary exceptions are the store-type instructions, where you can visualize data moving from the first operand to the second, or from left to right in the assembler instruction statement. For example:

```
ST    0,X                 c(GR0) —→ c(X)    Left to right
```

It also helps to remember how short operands are extended to the length of the target operand.

Operand Extension

When a source operand in a register or in memory is moved to a target register longer than the operand, the operand is extended to the length of the target register. Arithmetic loads extend the sign bit, and logical loads extend with zero bits.

Examples of arithmetic load instructions are LH, LGH, and LGFR; examples of logical load instructions are LLH, LLGH, and LLGFR.

We've seen a lot of new instructions in this section, and keeping track of them can be difficult. The following table provides a compact summary to help you understand how they are grouped and related.

Don't try to memorize!

The System z processors are very complex, and you'll learn the instruction mnemonics a few at a time. The tables at the end of each section summarizing the mnemonics and their opcodes are primarily for reference (and to help you in solving some of the Exercises).

Function	Oprnd1	8 bits	32 bits			64 bits				
	Oprnd2	8 bits	8 bits	16 bits	32 bits	8 bits	16 bits	31 bits	32 bits	64 bits
Load Arithmetic (from memory)			LB	LH	L LT LM	LGB	LGH		LGF LMH	LG LTG LMG
Load Arithmetic (from register)			LBR	LHR	LR LTR LPR LNR LCR	LGBR	LGHR		LGFR LTGFR LPGFR LNGFR LCGFR	LGR LTGR LPGR LNGR LCGR
Load Logical (from memory)			LLC	LLH		LLGC	LLGH	LLGT	LLGF	
Load Logical (from register)			LLCR	LLHR		LLGCR	LLGHR	LLGTR	LLGFR	
Store		STC		STH	ST STM STCM				STMH STCMH	STG STMG
Insert		IC			ICM ICMH					

Table 56. Summary of instructions discussed in this section

We'll use tables like this to summarize other instructions as they are introduced. (This one is more complex than most!)

In Table 56 you might say that the ICM/ICMH and STCM/STCMH instructions deal with one byte at a time; but because they might move up to 4 bytes, they are shown in the column with 32-bit second operands.

It is difficult to remember all these mnemonics, but they will become more familiar with regular use. You might ask why the System z architects didn't choose more descriptive mnemonics like `LoadRegister` and `InsertCharactersUnderMask`. Here are two reasons why not.

1. When you write Assembler Language programs, long mnemonics would require a lot of extra work that is saved by using short abbreviations.
2. In the early years of System/360, programs were prepared on 80-column punched cards, of which only 71 columns were available for Assembler Language statements. This meant that shorter mnemonics provided more space for name-field symbols, operands, and comments.

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
IC	43
ICM	BF
ICMH	EB80
L	58
LB	E376
LBR	B926
LCGFR	B913
LCGR	B903
LCR	13
LG	E304
LGB	E377
LGBR	B906
LGF	E314
LGFR	B914
LGH	E315
LGHR	B907
LGR	B904
LH	48
LHR	B927

Mnemonic	Opcode
LLC	E394
LLCR	B994
LLGC	E390
LLGCR	B984
LLGF	E316
LLGFR	B916
LLGH	E391
LLGHR	B985
LLGT	E317
LLGTR	B917
LLH	E395
LLHR	B995
LM	98
LMG	EB04
LMH	EB96
LNGFR	B911
LNGR	B901
LNR	11
LPGFR	B910

Mnemonic	Opcode
LPGR	B900
LPR	10
LR	18
LT	E312
LTG	E302
LTGFR	B912
LTGR	B902
LTR	12
ST	50
STC	42
STCM	BC
STCMH	EB2C
STG	E324
STH	40
STM	90
STMG	EB24
STMH	EB26

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
10	LPR
11	LNR
12	LTR
13	LCR
18	LR
40	STH
42	STC
43	IC
48	LH
50	ST
58	L
90	STM
98	LM
B900	LPGR
B901	LNGR
B902	LTGR
B903	LCGR
B904	LGR
B906	LGBR

Opcode	Mnemonic
B907	LGHR
B910	LPGFR
B911	LNGFR
B912	LTGFR
B913	LCGFR
B914	LGFR
B916	LLGFR
B917	LLGTR
B926	LBR
B927	LHR
B984	LLGCR
B985	LLGHR
B994	LLCR
B995	LLHR
BC	STCM
BF	ICM
E302	LTG
E304	LG
E312	LT

Opcode	Mnemonic
E314	LGF
E315	LGH
E316	LLGF
E317	LLGT
E324	STG
E376	LB
E377	LGB
E390	LLGC
E391	LLGH
E394	LLC
E395	LLH
EB04	LMG
EB24	STMG
EB26	STMH
EB2C	STCMH
EB80	ICMH
EB96	LMH

We will use tables like these to summarize instruction mnemonics and their operation codes as they are introduced.

Terms and Definitions

GR R_n
A notation referring to the rightmost 32 bits of the general register specified by R_n.

GR G_n
A notation referring to the full 64 bits of the general register specified by R_n.

GRn
A notation referring to the rightmost 32 bits of general register “n”.

GGn
A notation referring to 64-bit general register “n”.

insert
Place one or more bytes into a register without changing other bytes.

load operation
Replace the contents of a register with a copy of data from a memory address or from another register. Other parts of the register may contain sign-extended bits (for arithmetic loads), or zero-extended bits (for logical loads). The original contents of the target register are not preserved.

M_n
The field of a machine instruction designating a mask.

R_n
: The field of a machine instruction designating the number of a general register.

sign extension
: The process of making copies of the sign bit of a shorter operand and extending it to the left, to the length of a target field.

store operation
: Place a copy of part or all of a register's contents into memory.

zero extension
: The process of adding zero bits to the left of a shorter operand, to extend it to the length of a target field.

15. Testing the Condition Code: Conditional Branching

```
   11      55555555555
  111      55555555555
 1111      55
   11      55
   11      555555555
   11      55555555555
   11               555
   11                55
   11                55
   11               555
1111111111 55555555555
1111111111 555555555
```

Branch instructions let you choose alternative actions in your program, depending on tests or computed results whose status was indicated in the Condition Code.

The Condition Code is a two-bit field in the PSW (see Figure 12 on page 47), so its value is 0, 1, 2, or 3. To test the CC value we use a "Branch on Condition" instruction. The most common are the RX-type instruction BC and the RR-type instruction BCR. The result of testing the value of the CC determines whether or not the *branch condition* is met.

We'll start with the basic forms of conditional branch instructions; newer forms are discussed in Section 22. Other instructions whose actions depend on the value of the Condition Code are described later.

15.1. The Branch Address

If the condition for branching is *not* met (we'll see how to determine this in a moment), no action is taken and execution proceeds normally to the next sequential instruction following the Branch on Condition instruction.

If the branch condition *is* met, the *branch address* is determined:

1. For the BC instruction, the branch address is the Effective Address, determined from the displacement, base, and index fields of the instruction.
2. For the BCR instruction, the branch address is contained in the general register specified by the R_2 digit of the instruction. However, if the R_2 digit is zero, no branch ever occurs: that is, if R_2=0, the branch condition is never met.

To complete execution of a branch instruction, the IA portion of the PSW is *replaced* by the branch address. The next instruction to be fetched then comes from the address specified by the branch address. Branch instructions are also called "transfer" instructions, in the sense that control is transferred to the instruction at the branch address.

A successful branch instruction alters the normal sequencing of instruction fetching. If the IA is not changed by the branch instruction, the next instruction fetched follows the branch instruction, and we say that the branch was "not taken".

15.2. The Branch Mask and Branch Condition

The branch condition is determined by examining a *single* bit of the third hex digit of the instruction denoted "R_1" in Table 17 on page 107 and in Table 19 on page 108. For the BCR and BC instructions this digit does *not* refer to GR R_1, but is treated as a bit pattern called a *mask*, M_1, as we saw in Section 14.5 for the ICM and STCM instructions. The instructions have these formats:

07	M_1	R_2

Table 57. BCR instruction

47	M_1	X_2	B_2	D_2

Table 58. BC instruction

For both the RR and RX instructions, M_1 is the mask digit. Thus, we could write

```
        BCR   9,4               M1 = B'1001'
and
        BC    7,4(8,2)          M1 = B'0111'
```

Figure 79. Examples of conditional branch instructions

where the mask fields are `B'1001'` and `B'0111'` respectively.

The CPU matches the value of the CC to one of the mask bits, as shown in Table 59. If a 1-bit in the mask field position corresponds to the value of the CC, the branch condition *is* met; if the CC value matches a 0-bit in the mask, the branch condition is *not* met and no branch occurs.

CC value tested	**Instruction bit position**	**Mask bit position**	**Mask bit value**
0	8	0	8
1	9	1	4
2	10	2	2
3	11	3	1

Table 59. Mask bits and corresponding CC values

Thus in Figure 79, the `BCR 9,4` instruction would branch if the CC had values of 0 or 3, and the `BC 7,4(8,2)` instruction would not branch if the CC had value 0.

Exercises

15.2.1.(2) Show how the value of the CC can be used as a bit index in determining which bit of the mask digit to test.

15.2.2.(1)+ Why does a mask value of 15 imply an unconditional branch?

15.2.3.(1)+ What happens when `BC 15,0` is executed?

15.2.4.(3) If $0 \le n \le 15$, what will be the result of executing this instruction?

```
        BC    n,n(n,n)
```

15.2.5.(1)+ Using the information in Table 59, create a table with four rows of Condition Code values (0, 1, 2, and 3) and columns of 16 BC mask values that show at each intersection whether or not a branch will occur. (Feel free to transpose rows and columns if necessary.)

15.3. Examples of Conditional Branch Instructions

Here are some examples of conditional branching:

1. Branch to **XX** if the CC is zero.

```
         BC    8,XX                 M1 = B'1000'
```

   The BC instruction mask field has value B'1000', so the branch condition will be met only if the CC is zero.

2. Branch to **XX** if the CC is not 0.

```
         BC    7,XX                 M1 = B'0111'
```

   The mask has value B'0111', so the branch condition will be met if the CC is 1, 2, or 3.

3. Branch to the instruction whose address is contained in GR14.

```
         BCR   15,14                M1 = B'1111'
or
         BC    15,0(0,14)           M1 = B'1111'
```

   When all mask bits are one, the CC value *must* match a one-bit in the mask, so a branch always occurs: this is called an *unconditional branch*. We could also have written the BCR instruction as

```
         BCR   X'F',14
or
         BCR   B'1111',14           M1 is very clear here!
```

4. Branch to **XX** if the CC is 1 or 3.

```
         BC    5,XX                 M1 = B'0101'
```

15.4. No-Operation Instructions

We noted in example 3 above that a mask of all 1-bits means the branch is unconditional, because the branch condition is always met. Sometimes it is useful to execute an instruction that has *no* effect, so we usually use a conditional branch instruction with a zero mask field. Thus,

```
         BC    0,x
and
         BCR   0,any
```

have no effect, because the branch condition can never be met. They are sometimes called *"no-operation"* or *"no-op"* instructions, and the Assembler provides special "extended mnemonics" for them. The instructions

```
         NOP   S2
and
         NOPR  R2
```

are treated by the Assembler as being the same as

```
         BC    0,S2
and
         BCR   0,R2
```

respectively. Only a single operand is specified for each NOP or NOPR instruction, and the Assembler automatically provides the zero mask digit.

15.4.1. Special No-Operation Instructions (*)

One special type of no-operation instruction has an unusual side-effect. It has the form

```
         BCR   15,0                 M1 is B'1111'
```

Modern processors are highly "pipelined". That is, the fetch, decode, and execute phases are processed internally in many smaller steps called "stages".

Pipelining allows one instruction to begin its execution phase while the next is being decoded, and the instruction after that is being fetched.[97] Occasionally, it may be necessary to prevent this overlapped type of execution; this form of BCR instruction blocks execution of the following instruction until all preceding instructions have completed execution. This is sometimes called "draining" or "flushing" the pipeline, and it can cause programs to execute more slowly.

BCR operands can be interpreted this way:[98]

```
BCR   0,0      Branch never nowhere
BCR   15,0     Branch always nowhere (pipeline synchronization)
BCR   0,x      Branch never somewhere (when x > 0)
BCR   x,0      Branch sometimes nowhere (when 0 < x < X'F')
BCR   x,y      Branch sometimes somewhere (when 0 < x,y < X'F')
```

Exercises

15.4.1.(2)+ In trying to ensure that a BASR instruction was followed immediately by a word address constant, a programmer wrote the following instructions as part of his program:

```
DS    0F                  Align to fullword boundary
NOPR  0                   2-byte No-op
BASR  8,11                2-byte BASR instruction
DC    A(Anywhere)         Properly aligned fullword constant
```

Explain why this might create an unexpected problem.

15.4.2.(2) What other instructions could be used in place of NOPR and NOP?

15.4.3.(1) Suppose you execute the instruction

```
BCR   15,0
```

Will an unconditional branch occur? If so, from what address will the next instruction be fetched?

15.4.4.(2) Explain the execution-time differences between these two pairs of instructions:

```
L     1,=F'0'                 L     0,=F'0'
BCR   15,1                    BCR   15,0
```

15.5. Conditional No-Operation

An important use of "no-operation" instructions is to ensure a desired boundary alignment for a particular instruction in a stream of other executable instructions. (We have already seen how to obtain boundary alignments for *data.*) For example, we may require that an RR-type instruction such as

```
BASR  8,11
```

[97] The CPU is designed so that exception conditions at any stage are correctly recognized and handled as though each instruction is completely processed (or prevented from executing) before the next is fetched. Early pipelined processors couldn't always do this, and were subject to what were called "imprecise" interruptions. The CPU set the Instruction Length Code (ILC) to zero, meaning that both it (and you) weren't certain which instruction caused the interruption.

[98] This is not an official description.

be followed immediately (with no wasted space) by an aligned word constant such as an address constant. While it is best not to mix instructions and data this way, there are times where such a technique is useful.[99]

Since BASR is an RR-type instruction, we need to ensure that its location lies on a halfword boundary between two word boundaries. In a small program, it may be easy to determine the location of the BASR by counting LC values: if the BASR falls on a word boundary, insert a

```
         NOPR   0
```

instruction just before it. But if the program is large or if changes must be made somewhere preceding the BASR, it is difficult to know whether the NOPR should be inserted or not.

To do this automatically, the Assembler provides the CNOP (Conditional No-Operation) assembler instruction. If the LC is already on the desired boundary, nothing is inserted. Otherwise, CNOP inserts as many "NOPR 0" and "NOP 0" instructions as are needed to give the desired alignment.

The operand field entry of a CNOP instruction is written

```
         CNOP  boundary,width
```

where "boundary" and "width" are absolute expressions. The "boundary" operand may have any multiple-of-two value between 0 and 14, and its value must be less than the value of "width", which is 4, 8, or 16. A name field symbol is allowed, and its Length Attribute is always 1.

The "width" operand specifies the boundary relative to which alignment is performed, and "boundary" specifies the desired halfword relative to that boundary, as shown in Table 60.[100]

Instruction	Location Counter Alignment
CNOP 0,4	beginning of a word
CNOP 2,4	middle of a word
CNOP 0,8	beginning of a doubleword
CNOP 2,8	second halfword of a doubleword
CNOP 4,8	middle of a doubleword
CNOP 6,8	fourth halfword of a doubleword
CNOP 0,16	beginning of a quadword
CNOP 2,16	second halfword of a quadword
CNOP 4,16	second word of a quadword
CNOP 6,16	fourth halfword of a quadword
CNOP 8,16	second doubleword of a quadword
CNOP 10,16	sixth halfword of a quadword
CNOP 12,16	third word of a quadword
CNOP 14,16	eighth halfword of a quadword

Table 60. CNOP operands

To achieve the alignment desired for the BASR in our example, we would write

99 Modern CPUs maintain high-speed buffers known as "caches", one for fast access to instructions and another for fast access to data items. If data items appear in the instruction cache, the CPU must stop pre-processing instructions, load the data into the data cache (probably displacing useful data already there), and resume processing. This can cause significantly slower execution, so you should avoid "close" mixing of instructions and data.

100 More precisely,

```
         CNOP  boundary,width
```

causes the Assembler to insert enough "NOPR 0" or "NOP 0" instructions as may be needed to increment the LC (if necessary) so that the new value of the LC satisfies `boundary = LC (modulo width)`.

```
         CNOP  2,4                Align to middle of a word
         BASR  8,11               Two-byte instruction
         DC    A(AnyWhere)        No intervening bytes
```

Note that we should *not* write

```
         DS    0H
         BASR  8,11
         DC    A(AnyWhere)        No (??) intervening bytes
```

because alignment to a halfword boundary is *automatically* performed by the Assembler for instructions. Thus, the BASR could still fall on a word boundary, and the Assembler would then zero-fill the two bytes between the BASR and the address constant (because A-type constants have an implied word alignment). Similarly, we could not write

```
         BASR  8,11
         DS    0F
         DC    A(AnyWhere)
```

since the BASR could again fall on a word boundary, leaving two bytes between it and the constant that would be skipped by the Assembler. The contents of the two skipped bytes at execution time are arbitrary, since the Supervisor does not always clear or otherwise initialize the area into which a program is about to be loaded.

Name field symbols on CNOP instructions are rarely used, because branch-target symbols typically are given to instructions immediately preceding or following the CNOP. Thus, you could write a symbol that is the name of “nothing”:

```
         DS    0F                 Align on word boundary
CNopName CNOP  0,4                Align on word boundary (again?)
CallSub  BASR  14,15              Go to a subroutine
         - - -
```

The two symbols **CNopName** and **CallSub** will have the same LC value even though **CNopName** doesn't name anything different; it will have length attribute 1.

Figure 80 illustrates the alignment action of CNOP.

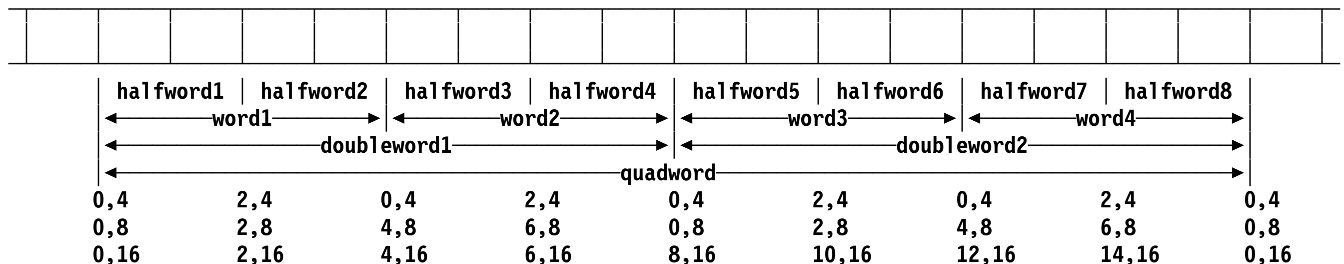


Figure 80. CNOP alignments and operands

Exercises

15.5.1.(2)+ Suppose the Location Counter value is `X'0246'` when each of these CNOP instructions is processed by the Assembler. Determine the value of the LC after each CNOP is processed.

```
(1)       CNOP  2,4
(2)       CNOP  0,4
(3)       CNOP  6,8
(4)       CNOP  10,16
(5)       CNOP  4,16
(6)       CNOP  2,8
```

15.5.2.(3)+ For each of these sets of statements the value of the Location Counter is X'000743' when the first statement is read by the Assembler. Give the length and value attributes of all symbols.

```
1. A      DC    AL3(A)
          CNOP  2,4
   B      DC    A(B)
2. C      DC    C'DS C''&'''
          CNOP  0,4
   D      DC    C'D DC D''DC'''
3.        CNOP  2,8
   E      DC    2F'100'
   F      DC    (X'F')X'F'
```

15.5.3.(1) In Exercise 15.5.2, what machine language data is generated by the statement named D?

15.5.4.(2)+ For each of the following, assume that the Location Counter value is X'345' when the initial statement is processed by the Assembler. Give the value and length attributes of the symbol A.

```
1.         CNOP  2,4
   A       LM    2,6,0(1)
2.         CNOP  2,8
   A       BC    10,Smith
```

15.5.5.(2)+What will be generated by a CNOP 6,8 statement?

15.6. Extended Mnemonics

Conditional branch instructions are used frequently. It can be difficult to remember the Condition Code values and mask bit values associated with possible branch conditions, so the Assembler provides *extended mnemonics* for conditional branch instructions. They let you *imply* the value of the mask field M_1 of a BC or BCR instruction by using an extended mnemonic. For example, an unconditional branch to an instruction named **XX** can be written

```
B    XX
```

which is easier and clearer than writing

```
BC   15,XX
```

Table 61 gives the extended mnemonics associated with the BC and BCR instructions. The notations "(A)", "(C)", and "(T)" refer to the contexts in which each extended mnemonic is most often used. The "A" mnemonics are typically used after Arithmetic instructions, "C" after Comparisons, and "T" after Tests.

Table 61 (Page 1 of 2). Extended branch mnemonics and their branch mask values

RX Mnemonic	RR Mnemonic	Mask	Meaning
B	BR	15	Unconditional Branch
BNO	BNOR	14	Branch if Not Ones (T) Branch if No Overflow (A)
BNH	BNHR	13	Branch if Not High (C)
BNP	BNPR	13	Branch if Not Plus (A)
BNL	BNLR	11	Branch if Not Low (C)
BNM	BNMR	11	Branch if Not Minus (A) Branch if Not Mixed (T)

Table 61 (Page 2 of 2). Extended branch mnemonics and their branch mask values			
RX Mnemonic	**RR Mnemonic**	**Mask**	**Meaning**
BE	BER	8	Branch if Equal (C)
BZ	BZR	8	Branch if Zero(s) (A,T)
BNE	BNER	7	Branch if Not Equal (C)
BNZ	BNZR	7	Branch if Not Zero (A,T)
BL	BLR	4	Branch if Low (C)
BM	BMR	4	Branch if Minus (A) Branch if Mixed (T)
BH	BHR	2	Branch if High (C)
BP	BPR	2	Branch if Plus (A)
BO	BOR	1	Branch if Ones (T) Branch if Overflow (A)
NOP	NOPR	0	No Operation

As this table indicates, the RR forms of the extended mnemonics are formed by adding the letter "R" to the equivalent RX mnemonic.

Each of these instructions needs only a single operand field entry. Because the mask digit is implied by the extended mnemonic, the operand may take any of the forms allowed for the second operand of an RX- or RR-type instruction.

For example, we could write example 1 of Section 15.3 on page 206 as

```
         BZ    XX
```

and example 2 could be written

```
         BNZ   XX
```

There is no extended mnemonic corresponding to a mask value of 5, so there is no convenient way to rewrite example 4.

Exercises

15.6.1.(2)+ Programmers sometime write programs that contain instruction sequences like this:

```
Loop     - - -                  Do something in the loop
         - - -                  Now make a test, set the CC
         BNZ   Finish           Exit the loop if something's nonzero
         B     Loop             Otherwise repeat the loop
Finish   - - -                  Rest of program
```

Why is this wasteful? How can it be made shorter, simpler, and (very probably) faster?

15.6.2.(1) Sometimes the conditional branch instructions are described as follows: "The operation code for an unconditional branch is X'47F', for a branch-on-zero is X'478', etc." Is this an accurate description?

15.6.3.(2) The word at **VAL** contains a 32-bit binary integer. Write an instruction sequence that will branch to **POS** if c(VAL) is greater than zero, to **NEG** if c(VAL) is less than zero, and to **ZERO** if c(VAL) is zero.

15.6.4.(2) A programmer accidentally wrote the operand of a branch instruction so that the branch target was a constant containing a string of space characters:

```
         B     Target            He meant to go elsewhere...
         - - -
Target   DC    CL132' '          132 bytes containing X'40'
```

What would you expect to happen?

15.7. A Comment on Programming Style

For short instruction sequences, it is sometimes tempting to avoid the effort of writing the name of a target symbol. For example, sometimes a programmer may write

```
LTR   0,0                       LTR   0,0
BZ    *+6    instead of         BZ    Next
LR    0,5                       LR    0,5
- - -                      Next - - -
```

The operand *+6 means the programmer knows that the lengths of BZ and LR are 4 and 2 bytes respectively, and this has saved him the “task” of writing the symbol **Next** in two places. However: suppose the logic of these statements needs to be updated, and extra instructions must be added following the BZ instruction. If the operand of BZ (which was not the cause of the change) isn't updated, it will branch into the added instructions and not to the intended target.[101]

A Programming Practice to Avoid

Do NOT write operands of branch instructions using the Location Counter value *± number.

Exercises

15.7.1.(2)+ A programmer wanted to be sure that the target of his branch instruction was at the correct location, so he wrote

```
         BZ    *+18              Branch to known target location
         - - -                   More instructions
*+18     EQU   *                 Define the target location.
```

Can he now be sure his BZ instruction will branch to the intended target?

15.8. A Design Oversight and a Modern “Correction” (*)

Due to a peculiarity in the original design of System/360 and System/370, invalid branch addresses were not detected during the execute phase of the instruction cycle at the time the CPU finds the branch condition is met. (Odd addresses produce specification errors, and excessively large addresses can produce addressing exceptions.) The error is found only when the bad address is presented, as the IA portion of the PSW, at the *next* instruction's fetch cycle. The error is duly detected and an interruption results, but the IA then contains the *invalid* address rather than the address of the instruction that attempted the improper branch. This means that looking at the “Old” PSW can't tell you where the error was caused, so such errors in a program are often very difficult to correct. You must specify branch addresses accurately to avoid this particular error.

The “Breaking Event Address Register” (sometimes called the “BEAR”) was added to z/Architecture. Whenever an instruction causes a break in normal sequential execution (such as a successful branch), the address of the instruction causing the “break” or discontinuity is placed in the BEAR. If the “break” causes a program interruption, the contents of the BEAR are stored in a fixed address where error detection and diagnosis routines can use the break address to help you find the instruction that caused the interruption.

Unfortunately, although the BEAR is accessible to ordinary problem-state programs, its contents aren't of much use unless its contents are captured at the moment an interruption occurs. So, to

[101] A clever programmer knew instruction lengths so well that he avoided writing name-field symbols on statements by coding instructions like B *+24 and BNZ *-20. Fixing errors in his code was *very* tedious.

answer questions like “How did my program start executing instructions *here*?”, we must depend on the operating system's Supervisor to save the BEAR's contents so the information can be used for problem diagnosis.

Exercises

15.8.1.(1)+ Explain why an odd branch address is invalid.

15.9. Summary

This section described the BC and BCR instructions and their forms as extended mnemonics. There are many other types of branch instructions, but their most important features are based on the concepts we've seen here; the others can be thought of as “variations” on the theme of this section. Newer forms of conditional branch instructions will be described in Section 22.1.

Exercises

15.9.1.(1) How could you design a CPU without a Condition Code or similar indicators?

15.9.2.(1)+ Can an instruction generate multiple CC values in a single execution?

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
BC	47

Mnemonic	Opcode
BCR	07

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
07	BCR

Opcode	Mnemonic
47	BC

Terms and Definitions

branch address
The address from which the next instruction will be fetched if the branch condition is met.

branch condition
The CPU's decision whether to alter the normal sequential execution of instructions by fetching instructions at the branch address.

branch mask
A 4-bit field in a Branch on Condition instruction used to test the value of the Condition Code. If a 1-bit in the branch mask matches the CC value, the branch condition is met.

conditional no-operation
An Assembler CNOP instruction that may generate NOP and NOPR instructions, causing the Location Counter to be aligned on a specified even boundary.

extended mnemonic
An instruction mnemonic provided by the Assembler allowing you to specify a branch mask implicitly.

no-operation instruction
An executable instruction having no effect other than to occupy space, to align a following instruction on a desired even boundary.

pipeline
A technique used in modern CPUs to speed instruction execution by dividing the fetch. decode, and execute phases into smaller stages that can be occupied by more than one instruction.

16. Fixed-Point Binary Addition, Subtraction, and Comparison

```
    11       6666666666
   111      666666666666
  1111      66        66
    11      66
    11      66
    11      66666666666
    11      666666666666
    11      66        66
    11      66        66
    11      66        66
1111111111  666666666666
1111111111   6666666666
```

This section describes instructions for fixed-point two's complement binary addition, subtraction, and comparison in the general registers, and between the general registers and memory. Because the instructions occur in very regular groups and patterns, understanding their basic behavior makes it easier to understand related instructions.

16.1. Signed-Arithmetic Add and Subtract Instructions

As we noted in Section 2.14 on page 37, logical addition and subtraction produce exactly the same bit patterns as arithmetic addition and subtraction, but the resulting CC settings have different meanings. We'll investigate logical-arithmetic instructions in Section 16.5 on page 224.

The instructions are shown in Table 62. The first six generate 32-bit results, and the others produce 64-bit results.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5A	A	RX	Add (32)	1A	AR	RR	Add Register (32)
5B	S	RX	Subtract (32)	1B	SR	RR	Subtract Register (32)
4A	AH	RX	Add Halfword (32←16)	4B	SH	RX	Subtract Halfword (32←16)
E308	AG	RXY	Add (64)	B308	AGR	RRE	Add Register (64)
E309	SG	RXY	Subtract (64)	B309	SGR	RRE	Subtract Register (64)

Table 62. Frequently used add and subtract instructions

16.2. Signed-Arithmetic Operations Using 32-Bit Registers

All the instructions in Table 62 on page 216 set the Condition Code as indicated in Table 63.

Operation	CC Setting and Meaning
$c(GR\ R_1) = c(GR\ R_1) \pm c(GR\ R_2)$ $c(GR\ R_1) = c(GR\ R_1) \pm c(\text{Word in memory})$	0: Result is zero; no overflow 1: Result is < zero; no overflow 2: Result is > zero; no overflow 3: Result has overflowed

Table 63. CC settings for arithmetic add and subtract instructions

Table 63 shows that these add and subtract instructions (like the LCR and LPR instructions in Table 44 on page 188) can cause a fixed-point overflow exception. We'll see in Section 16.9 on page 234 how to set the Program Mask to enable or disable an interruption.

We begin with the six add/subtract instructions A/AR, S/SR, and AH/SH. In each case, the second operand is added to or subtracted from the first operand, and the result replaces the first operand.

For the halfword operations AH and SH, the 16-bit second operand is brought from memory to an internal register, extended to a word in length, and then used for the indicated operation (as we saw in the description of LH in Section 14.3 on page 182). The notation (32←16) in Table 62 on page 216 means that the 16-bit operand is extended to 32 bits.

To illustrate arithmetic addition and subtraction, suppose we must store at **ANS** the sum of c(X) and c(Y), unless the sum is negative, in which case we must also add c(Z) and subtract 17. If we assume that **X**, **Y**, and **Z** name word areas of the program, then the following instructions will calculate the required value (assuming no overflows occur).

```
         L     6,X              Copy c(X) into GR6
         A     6,Y              c(GR6) = c(X) + c(Y)
         BNM   ST               Branch if sum is not negative
         A     6,Z              It was negative; add c(Z)
         SH    6,=H'17'         Subtract 17
ST       ST    6,ANS            Store answer at ANS
         - - -
X        DC    F'71'
Y        DC    F'-220'
Z        DC    F'284'
ANS      DS    F                Computed answer
```

Figure 81. Calculate a sum with an intermediate test

All the machine instructions are RX-type, and all but BNM refer to data operands in memory.

The characters “ST” are used both as a symbol and as an instruction mnemonic. No confusion is possible, since the Assembler identifies a mnemonic only by its appearance as an operation field entry.

Now, suppose we want to store the sum of the first N odd numbers at the word named **Sum** where the positive integer N is stored in the halfword integer at **NN**.

```
         LH    3,NN               Get the value of N from c(NN)
         LM    6,9,=F'0,2,1,1'    Load GR6-GR9 with 0,2,1,1
AddUp    AR    6,8                Add odd integer to sum in GR6
           AR    8,7              Next odd integer in GR8
           SR    3,9              Decrease N by 1
         BNZ   AddUp              Branch back (N-1) times
         ST    6,Sum              Store result in GR6 at SUM
         - - -
NN       DC    H'6'               Number of odd numbers to add
Sum      DS    F                  Sum of the first c(NN) odd numbers
```

Figure 82. Calculate the sum of the first N odd integers

In this example, the calculations inside the "loop"[102] (the AR and SR instructions beginning at **Addup**) are RR-type instructions; no memory references are needed. This technique is useful in programs where processing speed is important, and enough registers are available to allow frequently referenced operands to be carried in registers instead of in memory.[103]

To give another simple example of using some of these instructions, suppose we wish to compute the quantity **NewStock** from the formula

```
NewStock = OldStock + Received - Sold
```

where all quantities are word integers small enough in value to guarantee that no overflows will occur. These statements will compute the desired result.

```
         L    2,OldStock         Get c(OldStock) in GR2
         A    2,Received         Add number of items received
         S    2,Sold             Subtract number of items sold
         ST   2,NewStock         Store result at NewStock
```

Figure 83. Example of arithmetic addition and subtraction

Although we *assumed* that no overflows can occur, it may be possible that values calculated elsewhere in the program could cause an overflow here. Thus, to be careful, the above code sequence might continue with the instructions

```
         - - -                    (As above)
         ST    2,NewStock         (As above)
         BZ    ReOrder            None left, must order more!
         BM    OverSold           Reorder now! Sold more than stock!
         BO    Disaster           Error! More than 2**31 items??
```

Figure 84. Testing the result of arithmetic instructions

The instructions at **ReOrder** and **OverSold** will probably do similar things, except that at **OverSold** the order for new items would likely be given higher priority so our customers can receive their previously ordered products more promptly.

16.2.1. Condition Code Settings After Arithmetic

There is an important property of Condition Code settings after a binary addition and subtraction operation that causes an overflow: it is possible that the CPU could choose one of *two* CC settings. For example, adding `X'80000000'` to itself generates an overflow (the carry out of the sign bit isn't matched by a carry into the sign bit), and the arithmetic result is `X'00000000'`. The CPU follows this rule:

[102] A loop is a sequence of instructions executed repeatedly until some condition is satisfied. We'll see in Section 22 how other instructions help us write efficient loops.

[103] Don't be *too* impressed, however: the example is mathematically futile, because we have expended all this effort to calculate the square of N, when a single multiply instruction would have worked just as well. (See Exercise 16.2.2 for some mathematical background.)

Condition Code After Overflow

If a binary arithmetic operation causes fixed-point ovreflow, CC=3 is given preference to indicating any other property of the result.

That is, overflow indication is given priority.

See Exercise 16.2.16 and Programming Problem 16.12.

Exercises

16.2.1.(2) In Figure 81 on page 217, what will be stored at **ANS** if overflow occurs?

16.2.2.(3) Show that the sum of the first N odd integers is the same as N^2.

16.2.3.(2)+ In Figure 82 on page 218, we assumed that the positive integer N in c(NN) is 2 or greater. Rewrite the instructions to handle the possibility that N may be as small as 1.

16.2.4.(2)+ In a large program, a programmer wanted to decrement the value of an integer variable at **VARBL** by 1, so he wrote the instructions

```
Load     L    2,VARBL
         S    2,ONE
         ST   2,ONE     (Error! Meant to use 'VARBL'!)
         - - -
         B    Load      Try again
         - - -
VARBL    DC   F'4'
ONE      DC   F'1'
```

Unfortunately, the ST instruction stores the result in the wrong place! The program went into an infinite loop that included the three instructions above. What sequence of values appeared in the word named **ONE**?

16.2.5.(3)+ Given a word integer stored at **Data**, write an instruction sequence that will count the number of 1-bits in the word, and store the result in the halfword named **NBits**.

16.2.6.(3)+ Given a word integer stored at **Data**, write an instruction sequence that will determine the maximum power of 2 in the word, and store the result in the halfword named **MaxPow**. For example, the largest power of 2 in 9=B'1001' is 3. If c(Data) is zero, store −1, and if c(Data) is negative, store −31.

16.2.7.(2) In Figure 82 on page 218, what would happen if we had written

```
NN       DC    F'6'                ?
```

16.2.8.(3)+ Complete the “assembly” of this program segment by showing the generated object code and its locations.

```
 Loc   Object Code        Assembler Language Statements
                          Ex16_2_8 Start X'5000'
 5000  0D40________                BASR  4,0
 ____  _____________               Using *,4
 ____  _____________               SR    2,2
 ____  _____________               IC    2,XX+3
 ____  _____________               LTR   0,2
 ____  47___________               BZ    Looper
 ____  _____________               STH   0,XX
 ____  _____________      Looper   B     *       Loop forever here
 ____  _____________      YY       DC    CL4'Ugh'
 ____  _____________      XX       DC    F'-10'
```

16.2.9.(4) When the “program” in Exercise 16.2.8 is stopped (because it is in an unending loop), what will be the hexadecimal contents of the word at **XX**?

16.2.10.(2)+ Can an arithmetic operation using AH or SH cause fixed-point overflow? Explain.

16.2.11.(3)+ Suppose two constants are defined as follows:

```
X        DC    FL3'1234567'
Y        DC    FL3'7654321'
```

Write a sequence of instructions that will add the two numbers and store their sum as a 24-bit two's complement number in the three-byte field starting at W. If the sum overflows (it can't be represented correctly in 24 bits), branch to Over.

16.2.12.(2) In Exercise 16.2.11, what data is generated for the constant named Y?

16.2.13.(2) Explain the differences between these instruction pairs:

```
         SR    1,1            and            SR    0,0
         BCR   15,1                          BCR   15,0
```

Compare your answers to those you created for Exercise 15.4.4.

16.2.14.(3)+ Complete the “assembly” of this (nonsensical) program segment by showing the generated object code and its locations.

```
 Loc   Object Code         Assembler Language Statements
 8000                      Ex16_2_E Start X'8000'
 ____  _____________                BASR  4,0
 ____  _____________                Using *,4
 ____  _____________                LM    1,2,Value
 ____  _____________                STCM  2,B'111',First
 ____  _____________                LCR   0,1
 ____  _____________                BC    10,*+8
 ____  _____________                STH   0,Last(1)
 ____  _____________                BCR   15,14
 ____  _____________       Value    DC    F'4'
 ____  _____________                DC    F'-6'
 ____  _____________       First    DS    F
 ____  _____________       Last     DS    H'-10'
```

16.2.15.(2)+ Show the contents of GR2 and the Condition Code setting after executing each of the following instruction sequences:

```
 1.          L     2,=A(X'89ABCDEF')
             AR    2,2

 2.          L     2,=F'2'
             A     2,=A(X'7FFFFFFF')

 3.          L     2,=F'2'
             A     2,=A(X'123345')
             ICM   2,2,=X'12345'
```

16.2.16.(2)+ For each of these arithmetic operations, show the result and the CC setting.

```
1.       L     1,=X'80000000'
         S     1,=X'80000000'

2.       L     2,=X'00000000'
         S     2,=X'80000000'

3.       L     3,=X'FEDCBA98'
         A     3,=X'FEDCBA98'
```

```
4.        L     4,=X'FEDCBA98'
          S     4,=X'87654321'
```

16.3. Signed-Arithmetic Operations Using 64-Bit Registers

We now investigate the instructions in the second group pictured in Table 62 on page 216. The AG/AGR and SG/SGR instructions are 64-bit analogs of the 32-bit instructions A/AR and S/SR illustrated above. Condition Code settings are as shown in Table 63 on page 217. To illustrate, suppose we revise the example In Figure 81 on page 217 to use 64-bit operands:

```
         LG    6,XX
         AG    6,YY            c(GG6) = c(XX) + c(YY)
         BNM   ST              Branch if sum is not negative
         AG    6,ZZ            It was negative; add c(ZZ)
         SG    6,=FD'17'       Subtract 17 (doubleword literal)
ST       STG   6,DAnswer       Store result
         - - -
XX       DC    FD'7569241038'
YY       DC    FD'-94226701151'
ZZ       DC    FD'137'
DAnswer  DS    FD              Computed result
```

Figure 85. Calculate a 64-bit sum with an intermediate test

In this example, we cannot use the literal =H'17' because the System z instruction set does not (now) provide the AGH and SGH instructions.[104] (See Exercises 16.3.1 and 16.3.2.)

Suppose we add these two large numbers:

```
         LG    0,A             Get c(A)
         AG    0,B             ... and c(B)
         STG   0,C             Store sum at C
         - - -
A        DC    FD'9223372036854775807' = 2**63-1
B        DC    FD'9223372036854775807'
C        DS    FD              Result =X'FFFFFFFFFFFFFFFE' = -2, CC=3
```

Figure 86. Adding two 64-bit numbers

Because a fixed-point overflow has occurred, the result is arithmetically invalid.

Exercises

16.3.1.(2) Suppose you need to add a halfword value stored in memory at **HW** to a 64-bit value in GG0, and the CPU has no AGH instruction. What alternative instruction sequences could you use?

16.3.2.(2) Do the same as in Exercise 16.3.1, but now consider *subtracting* the **HW** operand from the 64-bit operand in GG0.

[104] At the time of this writing. But new instructions like AGHI (that we'll see in Section 21) are added regularly to the System z architecture, so check the *Principles of Operation*.

16.4. Signed-Arithmetic Compare Instructions

Table 64 lists the arithmetic compare instructions we'll examine:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
49	CH	RX	Compare Halfword (32←16)	E379	CHY	RXY	Compare Halfword (32←16)
59	C	RX	Compare (32)	19	CR	RR	Compare (32)
E359	CY	RXY	Compare (32)				
E320	CG	RXY	Compare (64)	B920	CGR	RRE	Compare (64)

Table 64. Arithmetic compare instructions

These instructions compare the magnitudes of two arithmetic operands. Thus, all positive numbers are greater than all negative numbers, and -2 is greater than -4. (We will see that logical comparisons behave differently.) The results of an arithmetic comparison are indicated in the CC setting, as shown in Table 65.

CC	Meaning
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

Table 65. CC settings after arithmetic comparisons

The CC cannot be set to 3 as a result of a compare instruction.

For the CR, C, and CH instructions, the CC setting is the same as would result from performing SR, S, and SH instructions with the same operands, assuming that no overflow occurs. In fact, this *is* how the comparison is done by the CPU: a subtraction is performed internally, and the CC is set to reflect the sign and magnitude of the difference (that would then have been placed back in GR R_1 or GG R_1 for a subtract instruction). Further analysis of the original operands is required by the CPU if the internal result overflows. (See Exercise 16.4.2.)

To illustrate arithmetic comparisons, consider these instructions and their comment fields:

```
         LM    0,3,=F'1,0,-1,-2147483647'  Initialize registers GR0-GR3
*        c(GR0) = -1, c(GR1) = 0, c(GR2) = +1, c(GR3) = X'80000001'
         CR    1,3         CC = 2             0 > -2147483647
         CR    0,2         CC = 2             1 > -1
         CR    2,3         CC = 2            -1 > -2147483647
         LPR   4,3         CC = 2   +2147483647 > 0
         CR    4,3         CC = 2   +2147483647 > -2147483647
         CR    1,0         CC = 1             0 < 1
         C     0,=F'1'     CC = 0             1 = 1
         CH    1,=H'5'     CC = 1             0 < 5
```

Figure 87. Examples of arithmetic comparisons

As an example of the use of a compare instruction, let us recalculate the sum of the first N odd integers, using a different scheme from the one in Figure 82 on page 218.

```
         LH    4,=H'1'       c(GR4) = accumulated sum
         LR    7,4           c(GR7) = count of additions
Test     CH    7,NN          Compare count to c(NN)
           BE    Store       Branch if equal, N terms added
           LR    0,7         Compute next odd integer
           AR    0,0         Counter + counter = 2N
           AH    0,=H'1'     Add 1, giving next odd term
           AR    4,0         Add term to sum
           AH    7,=H'1'     Increment count by 1
         B     Test          Branch back to see if finished
Store    ST    4,Sum         Store result
```

Figure 88. Calculate the sum of N odd integers

This example is cumbersome but yields the desired result.[105]

The arithmetic comparison instructions for 64-bit registers do exactly the same operations as the equivalent instructions do for 32-bit registers. If the second operand is shorter than the R_1 register, it is sign-extended internally to the length of the first operand before doing the comparison.

Exercises

16.4.1.(1)+ Why can the CC *not* be set to 3 in a comparison operation?

16.4.2.(3) In executing arithmetic compare instructions, the CPU performs an internal subtraction. By examining the possible combinations of signs and magnitudes for the two operands, determine (1) when an internal overflow might occur as a result of the internal subtraction, and (2) what the CPU must do to set the CC correctly in such cases.

16.4.3.(2) Suppose a programmer had written the last instruction in Figure 87 on page 222 as

```
         CH    1,=F'5'        (Rather than =H'5')
```

What would the CC setting be?

16.4.4.(2)+ In the following program, some pieces of data are missing, as indicated by the ____ spaces. Using the available information, fill in those spaces.

```
Loc   Object Code        Assembler Language Statements
                         Ex16_4_4 Start X'4800'
4800  _____________               BASR  10,0
4802  _____________               Using *,10
4802  _________A056      Loop     L     0,________
4806  _____________               A     0,One
480A  5000_________               ST    0,Number
480E   <other ops>                PrintOut Number
4824  59___________               C     0,Ten
4828  47___________               BL    Loop
482C   <other ops>                PrintOut *
4854       00000000      Number   DC    F'0'
4858       00000001      One      DC    F'1'
485C  _____________      Ten      DC    F'10'
                                  End   Ex16_4_4
```

[105] There are often many ways to perform the same computation. Programming is as much an art as a science, since you can write many different programs of varying degrees of efficiency, effectiveness, or elegance to achieve a given objective. A key consideration is that your program be understandable by others who may have to enhance (or fix) it in the future.

16.5. Logical-Arithmetic Add and Subtract Instructions

Logical-arithmetic instructions are used less often than signed-arithmetic instructions. They are typically used for extended-length or multiple-precision arithmetic (we'll see some examples), and on occasions when a sum or difference must be found without any possibility of a fixed-point overflow interruption. (The CPU calculates Effective Addresses using logical arithmetic, but does not set the Condition Code.)

Table 66 lists the logical arithmetic instructions we examine here:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5E	AL	RX	Add Logical (32)	1E	ALR	RR	Add Logical (32)
5F	SL	RX	Subtract Logical (32)	1F	SLR	RR	Subtract Logical (32)
E30A	ALG	RXY	Add Logical (64)	B90A	ALGR	RRE	Add Logical (64)
E30B	SLG	RXY	Subtract Logical (64)	B90B	SLGR	RRE	Subtract Logical (64)

Table 66. Logical arithmetic instructions

The CC settings we saw in Table 62 on page 216 for signed arithmetic are different for logical arithmetic. The Condition Code settings shown in Table 67 apply to all logical arithmetic instructions, so that references to c(GR R_1) also apply to c(GG R_1).

Operation	CC Setting and Meaning
c(GR R_1) = c(GR R_1) ± c(GR R_2) c(GR R_1) = c(GR R_1) ± c(Word in memory)	0: Zero result, no carry 1: Nonzero result, no carry 2: Zero result, carry 3: Nonzero result, carry
Note: CC0 cannot occur for logical subtraction	

Table 67. CC settings for logical add and subtract instructions

In Table 67, the CC settings for the logical arithmetic instructions depend only on whether a carry occurs out of the leftmost position of the R_1 register, and whether the result is zero. (Note that CC3 does *not* mean an overflow has occurred!) By referring to the examples in Sections 2.6 and 2.14, we see that the following rules hold:

1. A CC zero setting is possible for AL and ALR, and for ALG and ALGR, only if the first and second operands are both zero.
2. It is not possible to have a CC setting of zero for SL and SLR, or for SLG and SLGR. After the ones' complement of the second operand and a low-order 1-bit are added to the first operand, a carry must have occurred if the result is zero.

To illustrate the differences between arithmetic and logical addition and subtraction, consider examples 1 and 2 of Section 2.11 on page 32.

- Example 1. For unsigned operands, the result of 5−3=2 is representable.

```
                     5-3:        0000 0101
                                -0000 0011
        becomes
                                 0000 0101
                                +1111 1101
                (carry lost)     0000 0010 = 2
```

  When we logically subtract unsigned operands, the presence of a carry means that the result was valid, and that there was no need to “borrow” from any higher-order digit positions.

- Example 2. For unsigned operands, the result of 3−5 cannot be correctly represented without “borrowing” from higher-order digit positions (negative values don't exist in this 8-bit representation).

```
                3-5:          0000 0011
                             -0000 0101
     becomes
                              0000 0011
                             +1111 1011
           (no carry)         1111 1110 = -2 (arithmetically, not logically!)
```

Thus, when logically subtracting unsigned operands, the absence of a carry means that we need to "borrow" from a higher-order digit position.

Table 68 summarizes these observations:

Operation	Carry	No Carry
Logical Addition	Carry to higher-order position	No carry to higher-order position
Logical Subtraction	No borrow from higher-order position	Borrow from higher-order position

Table 68. CC indications for logical addition and subtraction

As in Figure 86 on page 221, we can use logical arithmetic to add the same two numbers:

```
         LG    0,A                Get c(A)
         ALG   0,B                ... and c(B)
         STG   0,C                Store sum at C
         - - -
A        DC    FD'9223372036854775807'
B        DC    FD'9223372036854775807'
C        DS    FD                 Result =X'FFFFFFFFFFFFFFFE', CC=3
```

Figure 89. Adding two 64-bit numbers logically

The result at **C** is the same as before, but now there is no fixed-point overflow.

In the next section we will see how the presence or absence of a carry condition is used when we add and subtract "long" or "multiple-precision" numbers.

To illustrate a typical use of logical arithmetic, suppose we must add and subtract 64-bit integers represented by pairs of 32-bit integers: that is, double-length integers two words long. (Double-length integers are also encountered as products and dividends.) That is, we must do integer arithmetic with operands longer than a single general register.

First, consider how we find the two's complement (the negative) of such a 64-bit number. Since we know that the two's complement is found by adding a low-order 1-bit to the *ones'* complement of the number, we might proceed as follows. The number to be complemented is stored in a doubleword at **ARG**, and c(GR0,GR1) means the contents of the double-length register pair formed by GR0 and GR1.

```
         L     0,=F'-1'           All one bits in GR0
         LR    1,0                c(GR0,GR1) is now all 1-bits
         SL    0,ARG              Ones' complement of high-order part
         SL    1,ARG+4            Ones' complement of low-order part
         AL    1,=F'1'            Now add the low-order 1 bit
         BC    B'1100',NoCarry    Branch if no carry out of GR1 occurs
         AL    0,=F'1'            Propagate the carry bit to GR0
NoCarry  STM   0,1,ARG            Store final result back at ARG
         - - -
ARG      DC    FD'123456787654321'   64-bit integer
```

Figure 90. Double-length complementation

The first AL instruction must be used rather than an A instruction because the high-order bit of GR1 is not a sign bit, but an arithmetically significant bit with weight 2^{31}. If a carry out of GR1 occurs, it must be detected and propagated into the low-order bit of GR0.

The same complementation is performed by the following code sequence, but more directly (and less obviously).

```
         LM    0,1,ARG            Get double-length operand
         LCR   0,0                Complement high-order half
         LCR   1,1                Complement low-order half
         BZ    XXX                Jump if c(GR1) was 0
         SL    0,=F'1'            Subtract 1 from GR0
XXX      STM   0,1,ARG            Store result at ARG
         - - -
ARG      DC    FD'123456787654321'   64-bit integer
```

Figure 91. Double-length complementation, a simpler way

The first LCR instruction forms the two's complement of the high-order 32 bits in c(GR0); that is, we have already added a low-order 1-bit to the ones' complement of c(GR0). The following LCR complements the low-order 32 bits, and sets the CC. If c(GR1) had been zero, its ones' complement would have been all 1-bits, and adding a low-order one would cause a carry out the left end of R1; the first LCR has already “propagated” a carry into GR0. For *any other* bit pattern, no such carry would have occurred, so we must correct c(GR0) by subtracting off the low-order bit that was automatically added during the execution of the first LCR.[106]

Adding the two double-length integers at **A** and **B** is straightforward: the instructions are explained in the comments.

```
         LM    0,1,A              Load A into c(GR0,GR1)
         AL    1,B+4              Add low-order part of B
         BC    B'1100',NoCarry    Branch if no carry
         AL    0,=F'1'            Propagate carry to high-order word
NoCarry  AL    0,B                Add high-order part of B
         STM   0,1,Sum            Store the double-length sum
         - - -
Sum      DS    FD                 8 bytes, aligned
A        DC    FD'888777666555'
B        DC    FD'222333444555'
```

Figure 92. Double-length addition

Subtracting 64-bit operands is done the same way, except that the condition code setting after the first logical subtraction requires explanation.

```
         LM    0,1,A              Get first operand as c(GR0,GR1)
         SL    1,B+4              Subtract low-order parts
         BC    B'0011',NoBorrow   Branch if there's a carry
         SL    0,=F'1'            Reduce c(GR0) by 1 (i.e., borrow 1)
NoBorrow SL    0,B                Subtract high-order parts
         STM   0,1,Diff           Store 64-bit difference
         - - -
Diff     DS   FD
A        DC   FD'234567898765432'
B        DC   FD'123456787654321'
```

Figure 93. Double-length subtraction

[106] This instruction sequence has a minor defect: if either of the LCR instructions complements the maximum negative number `X'80000000'`, a fixed-point overflow exception could occur. (See Exercise 16.6.5.)

In performing a subtraction, the ones' complement of the second operand and a low-order 1-bit are added to the first operand. If a carry occurs out of the high-order bit position of the low-order register, then the result is correctly represented. If a carry does not occur the result is not correctly represented, in the sense that we have tried to generate a “negative” integer in the logical representation. Hence we must “borrow” a 1-bit from the next higher bit position, so we subtract =F'1' if the branch condition is not met.

It might help to review the examples in Section 2.11 on page 32 to clarify the relationship between carries and overflow in the arithmetic and logical representations. The instructions in Section 16.6 greatly simplify these operations.

Using 32-bit registers to calculate 64-bit results is unnecessary if you need only 64-bit results, because you can use 64-bit operations instead. But if you need to calculate the 128-bit sum of two 64-bit operands, these techniques are useful. (See Exercise 16.7.4.)

To see how logical arithmetic can provide possibly misleading arithmetic results, consider the example in Figure 83 on page 218, revised to use logical add and subtract instructions:

```
L    2,OldStock          Get c(OldStock) in GR2
AL   2,Received          Add number of items received
SL   2,Sold              Subtract number of items sold
ST   2,NewStock          Store result at NewStock
```

Figure 94. Example of logical addition and subtraction

These instructions (using logical add and subtract) are not recommended, for two reasons. First, although the result stored at **NewStock** is the same in both cases, the CC setting is not; if we follow the ST instruction by conditional branch instructions that depend on the arithmetic sign of the result (as in Figure 84 on page 218), the branch instructions may not go to the intended targets.

Exercises

16.5.1.(2) Suppose the instruction sequence in Figure 94 is followed by the three branch instructions in Figure 84 on page 218. What results will cause branching to each of the three target symbols?

16.5.2.(2)+ In the complementation instructions shown in Figures 90 and 91, what additional instructions would be needed to cause a branch to **OverFlow** if the 64-bit result of the complementation overflowed?

16.5.3.(3)+ In the addition instructions shown in Figure 92 on page 226, what additional instructions would be needed to cause a branch to **OverFlow** if the 64-bit result of the addition overflowed?

16.5.4.(2)+ In the subtraction instructions shown in Figure 93 on page 226, what additional instructions would be needed to cause a branch to **OverFlow** if the 64-bit result of the subtraction overflowed?

16.5.5.(2) In Figure 91 on page 226, if either 32-bit operand is the maximum negative number, complementation by the LCR instructions will cause a fixed-point overflow condition. Revise the instructions to produce the 64-bit two's complement without any overflow condition.

16.5.6.(3) Examine the instructions in Figures 92 and 93. Make a short table indicating all the possible CC settings, and the operands that produce them.

16.5.7.(3) Examine the instructions in Figures 92 and 93. Revise them to set the contents of the word at **CCode** to contain the correct CC setting after addition and subtraction. If you can make the *actual* CC setting correct, so much the better.

16.5.8.(3) Write a sequence of instructions that form the two's complement of a 64-bit integer represented as a pair of 32-bit words, that also set the CC to the same value as LCGR does for the same 64-bit integer.

16.5.9.(3) In the examples of the addition and subtraction of double-length numbers in Figures 92 and 93, make modifications to the code such that if the final double-length result overflows, control will be transferred to **OVER**. The register contents need not be correct if such a transfer is made.

16.5.10.(4) Do the same as for Exercise 16.5.9, but after the addition or subtraction, the word named **CCode** should reflect the condition of the double-length result, which should also be correctly represented to 64 bits. That is, using 32-bit registers, compute the 64-bit sum as though a 64-bit addition is performed. Extra credit: make the *actual* CC setting correct,

16.5.11.(2)+ For the logical add and subtract instructions, each bit of the CC has a particular meaning. Make a table with two rows and two columns summarizing the meanings of the four possible CC values as a function of the values of its two bits.

16.5.12.(1) If a logical subtraction is performed with two operands that are identically zero, why is the resulting CC setting not zero?

16.6. Add With Carry, Subtract With Borrow (*)

Referring to Table 67 on page 224, we can represent the Condition Code settings for logical *addition* in a different way, as shown in Table 69.

CC bit	0	1
Left	No carry	Carry
Right	Zero result	Nonzero result

Table 69. CC settings after logical addition

Thus, the leftmost bit of the CC can be thought of as the "carry bit". Similarly, referring to Table 68 on page 225, another way to represent the CC settings for logical *subtraction* is provided in Table 70.

CC bit	0	1
Left	Borrow (no carry)	No borrow (carry)
Right	Zero result	Nonzero result

Table 70. CC settings after logical subtraction

The instructions in Table 71 take advantage of the leftmost CC bit to minimize the number of instructions needed to do double-length (or multiple-length) arithmetic[107] by using the CC bit to propagate a carry or borrow to the next higher-order operand.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E398	ALC	RXY	Add Logical with Carry (32)	B998	ALCR	RRE	Add Logical with Carry (32)
E388	ALCG	RXY	Add Logical with Carry (64)	B988	ALCGR	RRE	Add Logical with Carry (64)
E399	SLB	RXY	Subtract Logical with Borrow (32)	B999	SLBR	RRE	Subtract Logical with Borrow (32)
E389	SLBG	RXY	Subtract Logical with Borrow (64)	B989	SLBGR	RRE	Subtract Logical with Borrow (64)

Table 71. Logical arithmetic instructions with carry/borrow

107 Multiple-precision arithmetic is used intensively in cryptographic applications for data security.

Now, we can use these instructions to improve the examples of double-length addition and subtraction shown in Figures 92 and 93 on page 226. First, consider addition: now, the intermediate branch and addition of a low-order 1 are unneeded.

```
         LM    0,1,A             Load A in register pair
         AL    1,B+4             Add low-order part of B
         ALC   0,B               Add high-order part of B with carry
         STM   0,1,Sum           Store the double-length sum
         - - -
Sum      DS    FD                8 bytes, aligned
A        DC    FD'888777666555'
B        DC    FD'222333444555'
```

Figure 95. Double-length addition with carry

Similarly, the double-length subtraction can be rewritten:

```
         LM    0,1,A             Get first operand
         SL    1,B+4             Subtract low-order parts
         SLB   0,B               Subtract high-order parts with borrow
         STM   0,1,Diff          Store 64-bit difference
         - - -
Diff     DS   FD
A        DC   FD'234567898765432'
B        DC   FD'123456787654321'
```

Figure 96. Double-length subtraction with borrow

Exercises

16.6.1.(2)+ Repeat Exercise 16.5.9, using Add Logical With Carry and Subtract Logical With Borrow instructions as appropriate.

16.6.2.(3) Repeat Exercise 16.5.9, using Add Logical With Carry and Subtract Logical With Borrow instructions as appropriate, this time storing the proper Condition Code value at **CCode**.

16.6.3.(3)+ Suppose two 256-bit integers are stored as eight consecutive words (or four consecutive doublewords) in memory starting at **A256** and **B256** respectively. Using Add Logical With Carry and Subtract Logical With Borrow instructions, write instructions to store their sum and difference at **Sum256** and **Diff256** respectively.

16.6.4.(3) In Exercise 16.6.3, the add and subtract instructions do logical arithmetic. How would you detect an arithmetic overflow?

16.6.5.(2)+ Write an instruction sequence using ALC to add two 128-bit numbers represented as two groups of four fullwords each.

16.7. Operations With Mixed 64-Bit and 32-Bit Operands

The instructions in Table 72 on page 230 all involve a 64-bit first operand and a 32-bit second operand.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B318	AGF	RXY	Add (64←32)	B318	AGFR	RRE	Add Register (64←32)
B309	SGF	RXY	Subtract (64←32)	B319	SGFR	RRE	Subtract Register (64←32)
E31A	ALGF	RXY	Add Logical (64←32)	B91A	ALGFR	RRE	Add Logical (64←32)
E31B	SLGF	RXY	Subtract Logical (64←32)	B91B	SLGFR	RRE	Subtract Logical (64←32)
E330	CGF	RXY	Compare (64←32)	B930	CGFR	RRE	Compare (64←32)

Table 72. Instructions for mixed-length operands

The AGF and SGF instructions are similar to AH and SH, except that instead of sign-extending a 16-bit memory operand to 32 bits, a 32-bit memory operand is extended to 64 bits before participating in the 64-bit operation, as illustrated in Figure 97.

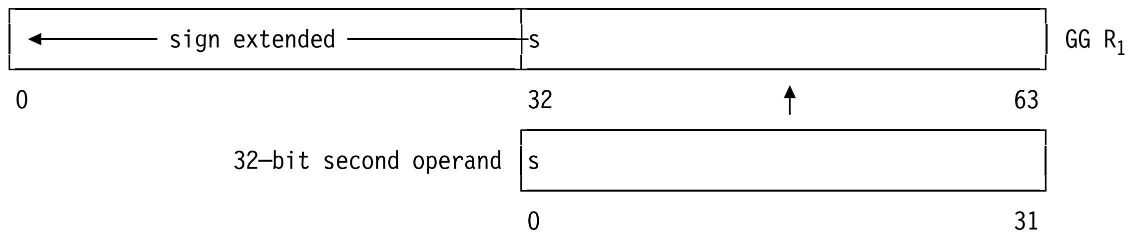


Figure 97. Sign extension for instructions with mixed 32- and 64-bit signed operands

Using SGF, we can modify the example in Figure 85 on page 221 to use a word literal:

```
         LG    6,XX
         AG    6,YY              c(GG6) = c(XX) + c(YY)
         BNM   ST                Branch if sum is not negative
         AG    6,ZZ              It was negative; add c(ZZ)
         SGF   6,=F'17'          Subtract 17 (word literal)
ST       STG   6,DAnswer         Store result
         - - -                   etc.
```

Figure 98. Calculate a 64-bit sum with an intermediate test

The AGFR and SGFR instructions use the same sign-extension process for 32-bit second operands in general registers as AGF and SGF do for 32-bit second operands in memory. For example, if we must use only a halfword operand such as =H'17', we can rewrite Figure 98 as follows:

```
         LG    6,XX
         AG    6,YY              c(GG6) = c(XX) + c(YY)
         BNM   ST                Branch if sum is not negative
         AG    6,ZZ              It was negative; add c(ZZ)
         LH    0,=H'17'          Load 17 into GR0 (32 bits)
         SGFR  6,0               Extend; then subtract GR0 from GG6
ST       STG   6,DAnswer         Store result
         - - -                   etc.
```

Figure 99. Calculate a 64-bit sum with an intermediate test

This approach requires an additional register (GR0) as a “temporary” register, which may be inconvenient. Figure 99 is also one instruction and two bytes longer (counting the literal) than Figure 98, so we could have used a word operand such as =F'17'.

Because logical arithmetic uses unsigned nonnegative operands, all bits have positive weight. Thus, when an instruction requires unsigned operands with mixed lengths, the shorter operand is always “sign-extended” with zero bits, as shown in Figure 100 on page 231.

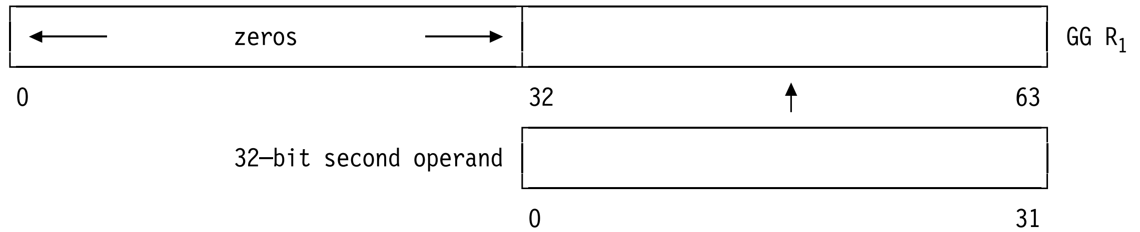


Figure 100. Sign extension for instructions with mixed 32- and 64-bit unsigned operands

For example:

```
(1)      LG    0,=AD(X'0123456789ABCDEF')
         ALG   0,=AD(X'123456789ABCDEF0') c(GG0)=X'13579BE02468ACDF', CC=1
```

Adding the two operands causes no overflow and the result is nonzero, so CC=1.

```
(2)      LG    0,=AD(X'0123456789ABCDEF')
         ALGF  0,=A(X'87654321')          c(GG0)=X'0123456811111110', CC=1
```

As in (1), but the second operand is first extended with zeros.

```
(3)      LG    0,=AD(X'0123456789ABCDEF')
         SLGF  0,=A(X'87654321')          c(GG0)=X'0123456702468ACE', CC=3
```

Subtracting the second operand causes a carry and the result is nonzero, so CC=3.

```
(4)      SR    1,1                 c(GR1)=0
         SGR   0,0                 c(GG0)=0
         SLGFR 0,1                 c(GG0)=X'0000000000000000', CC=2
```

Subtracting the second operand causes a carry and the result is zero, so CC=2.

Exercises

16.7.1.(2) Revise the instructions shown in Figure 91 on page 226 to complement a pair of 64-bit integers, giving a 128-bit result.

16.7.2.(2) Revise the instructions shown in Figure 92 on page 226 to add a pair of 64-bit integers, giving a 128-bit sum.

16.7.3.(2) Revise the instructions shown in Figure 93 on page 226 to subtract a pair of 64-bit integers, giving a 128-bit difference.

16.7.4.(3) Write instructions to form the 128-bit sum and difference of the pair of 64-bit integers stored starting at **Two64s**. Store the sum at **Sum128** and the difference at **Diff128**.

16.7.5.(1) Show the CC values after executing

```
         SLR   0,0
and
         SLGR  0,0
```

and after executing

```
         SR    0,0
and
         SGR   0,0
```

16.8. Logical-Arithmetic Compare Instructions

The logical compare instructions are shown in Table 73.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
55	CL	RX	Compare Logical (32)	15	CLR	RR	Compare Logical (32)
E321	CLG	RXY	Compare Logical (64)	B921	CLGR	RRE	Compare Logical (64)
E331	CLGF	RXY	Compare Logical (64←32)	B931	CLGFR	RRE	Compare Logical (64←32)
BD	CLM	RS	Compare Logical Characters under Mask (32)	EB20	CLMH	RSY	Compare Logical Characters under Mask (32)
EB21	CLMY	RSY	Compare Logical Characters under Mask (32)				

Table 73. Arithmetic compare instructions

As we saw in Section 14.7 on page 189, RXY- and RSY-type instructions behave the same way as RX- and RS-type instructions.

The logical compare instructions test the relative magnitudes of two operands, using an unsigned comparison instead of the signed-arithmetic comparison used for arithmetic comparisons. The results of all logical comparisons are indicated in the CC setting, as shown in Table 74 (you'll note that it's identical to Table 65 on page 222).

CC	Meaning
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

Table 74. CC settings after logical comparisons

Logical comparisons do *not* give the same results as arithmetic comparisons, since numbers in the logical representation are always nonnegative. The following instruction sequence may help to show the differences. (Following the LM instruction, the contents of R3 will be X'80000001'.)

The 64-bit logical comparison instructions behave the same way as their 32-bit equivalents. Carefully compare the CC settings in Figure 101 with those in Figure 87 on page 222.

```
LM    0,3,=F'1,0,-1,-2147483647'  Initialize registers GR0-GR3
CLR   1,3          CC = 1    X'00000000' < X'80000001'
CLR   0,2          CC = 1    X'00000001' < X'FFFFFFFF'
CLR   2,3          CC = 2    X'FFFFFFFF' > X'80000001'
LPR   4,3          CC = 2;  (now, c(GR4) = X'7FFFFFFF')
CLR   4,3          CC = 1    X'7FFFFFFF' < X'80000000'
CL    2,=F'+2'     CC = 2    X'FFFFFFFF' > X'00000002'
CH    1,=H'5'      CC = 1    X'00000000' < X'00000005'
```

Figure 101. Examples of logical comparisons

The CLM and CLMH instructions are unlike the other compare instructions, because the entire first operand might not be used. Instead, they operate on selected bytes in the register, as determined by 1-bits in the M_3 mask field of the instruction (just as we saw for the ICM/ICMH instructions in Section 14.5 on page 185). The selected bytes in the register are compared to the string of bytes in memory beginning at the second operand address. The comparison is performed by considering the two strings to be unsigned logical numbers of length 8, 16, 24, or 32 bits. If the mask digit M_3 is zero, the CC is set to zero and no comparison is performed.

- CLM and CLMY compare selected bytes in the first operand register (either in a 32-bit register or in the rightmost 32 bits of a 64-bit register) to the storage operand. For example:

```
L     0,=A(X'00010203')    Initialize GR0
CLM   0,B'0000',=X'0123'   CC = 0, because mask is 0
CLM   0,B'0001',=X'0123'   CC = 2, because X'03' > X'01'
CLM   0,B'0100',=X'0123'   CC = 0, because X'01' = X'01'
CLM   0,B'0110',=X'0123'   CC = 1, because X'0102' < X'0123'
```

- CLMH does exactly the same as CLM, except that it compares bytes in the high-order half of a 64-bit register to bytes in memory. For example:

```
LG    0,=AD(X'0001020304050607')    Initialize GG0
CLMH  0,B'0000',=X'0123'   CC = 0, because mask is 0
CLMH  0,B'0001',=X'0123'   CC = 2, because X'03' > X'01'
CLMH  0,B'0100',=X'0123'   CC = 0, because X'01' = X'01'
CLMH  0,B'0110',=X'0123'   CC = 1, because X'0102' < X'0123'
```

  The bytes in the low-order half of the 64-bit register are ignored.

Sometimes the logical compare instructions are used to test the ordering of values that are regularly incremented. For example, if **Value** has been saved at different times, we could find that

```
Oldest   DC    X'789ABCDE'        Oldest value
Later    DC    X'89ABCDEF'        A later value
Newest   DC    X'9ABCDEF0'        Most recent value
```

and we can use logical comparisons to determine their ordering, as in

```
L     0,Oldest
L     1,Later
L     2,Newest
- - -
CLR   1,0                Compare Later to Oldest
CLR   2,1                Compare Newest to Later
```

Figure 102. Comparing logically ordered values

then both CLR instructions will give the correct ordering of the three values.

But if the values can "wrap around" from X'FFFFFFFF' to zero, we must be more careful. For example, suppose the three values are

```
Oldest   DC    X'FFFFFFFE'        Oldest value
Later    DC    X'FFFFFFFF'        A later value
Newest   DC    X'00000001'        Most recent value
```

Then if we compare them as previously, the second comparison will fail, because the value at **Newest** will be logically less than the value at **Later**.

To avoid this problem, we can write instead

```
LR    3,1                Copy Later value to GR3
SLR   3,0                Subtract Oldest value
LTR   3,3                Test result
```

and the Condition Code will indicate that c(Later) is indeed greater than c(Oldest). Similarly, if we write

```
LR    3,2                Copy Newest value to GR3
SLR   3,1                Subtract Later value
LTR   3,3                Test result
```

the CC will again indicate the correct ordering.

Exercises

16.8.1.(2) Show how the CC settings after SL and SLR are related to those after CL and CLR.

16.8.2.(2) Suppose GG0 contains X'1122334455667788', and you must compare bytes 2 through 5 (containing X'33445566') to a 4-byte memory operand named **StgOp**. Write an instruction or sequence of instructions to do this.

16.8.3.(2)+ Suppose c(GR0) is X'87654321' and c(GR1) is X'01234567'. What is the CC setting and the apparent ordering of the operands after executing each of these two instructions?

```
CR    0,1                Compare c(GR0) to c(GR1)
CLR   0,1                Compare c(GR0) to c(GR1)
```

Now, suppose the sign bit of each operand has been inverted, so that c(GR0)=X'07654321' and c(GR1)=X'81234567'. What is the CC setting and the apparent ordering of the operands after executing each of the two instructions? Why might this sign-bit inversion be useful?

16.8.4.(2) Make a table showing the first and second comparison operands in Figures 87 and 101, and the CC settings from their arithmetic and logical comparisons. For which operands are they the same, and why?

16.8.5.(2) What differences will occur if two binary numbers are compared using arithmetic and then logical compare instructions?

16.8.6.(2)+ Write and execute a small program to verify the assertions about correctly-ordered logical comparisons in the examples starting with Figure 102 on page 233.

16.9. Retrieving and Setting the Program Mask (*)

The IPM and SPM instructions in Table 75 let you retrieve and set the value of the Condition Code and the Program Mask (PM).

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B222	IPM	RRE	Insert Program Mask	04	SPM	RR	Set Program Mask

Table 75. IPM and SPM instructions

Both instructions have a single operand:

```
IPM   R1          Insert CC and Program Mask into GR R1
SPM   R1          Set CC and Program Mask from GR R1
```

IPM inserts the Condition Code and Program Mask into bits 34-39 of register R_1, in the positions shown in Figure 103; the remaining bits of the R_1 register are unchanged. Conversely, SPM sets the Condition Code (CC) and Program Mask from the same bit positions, and ignores the rest of the R_1 register.

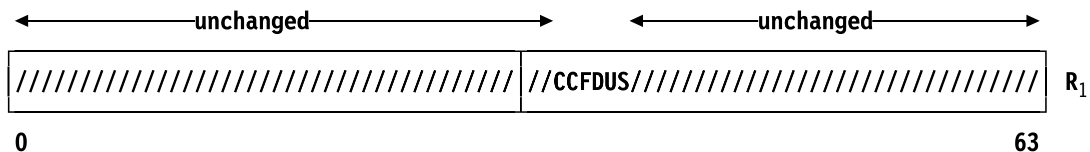


Figure 103. Bit positions used by IPM and SPM instructions (System/360 PSW sketch)

The four mask bits in the Program Mask ("**FDUS**" in Figure 103) control the behavior of the four exceptions described in Section 4.6 on page 55. These four mask bits correspond to the bit positions shown in Table 76 on page 235:

Bit	Exception Condition Controlled	Int. Code
36 (F)	Fixed-point overflow	8
37 (D)	Decimal overflow	A
38 (U)	Hexadecimal floating-point underflow	D
39 (S)	Hexadecimal floating-point lost significance	E

Table 76. Program Mask bits

Setting a mask bit to 1 *enables* the corresponding interruption. If the mask bit is 0, the CPU takes a default action without an interruption.

In practice, many programmers choose to set the Program Mask to zero initially, and trust to luck that nothing goes wrong. For example:

```
SR    0,0                Set c(GR0) to zero
SPM   0                  Set CC and Program Mask bits to zero
```

Careful placement of tests for overflow can help justify such faith, but it is generally better to test in advance for possible errors, and let a program interruption catch the unexpected and truly exceptional cases.

For now, we are concerned only with fixed-point overflow. The result of an instruction causing a fixed-point overflow is the same whether or not an interruption occurs; the Condition Code is set to 3.

Exercises

16.9.1.(4) For each of the conditions controlled by a bit in the Program Mask, determine what actions are taken by the CPU (including CC settings) when the PM bit is zero or one. (You may need to consult the *z/Architecture Principles of Operation*.)

16.9.2.(2)+ Write instructions that will turn off the Lost-Significance mask bit in the Program Mask, without affecting the settings of the other mask bits.

16.9.3.(2) Assume you are executing in 24-bit addressing mode. The fullword integer at **CCode** has a value of 0, 1, 2, or 3. Set the Condition Code to that value, without affecting the setting of the Program Mask.

16.9.4.(2) Assume you are executing in 24-bit addressing mode. Store the current value of the program mask in the rightmost four bits of the byte at **PMask**. The remaining 4 bits of the byte should be zero.

16.9.5.(2) Assume you are executing in 24-bit addressing mode. Store the current value of the Condition Code in the word at **CCode** without changing the Condition Code.

16.10. Summary

Operands used in arithmetic and logical operations may be extended, as we noted in Sections 14.10 and 16.7.

Operand Extension

When a source operand in a register or in memory is used as an operand in an arithmetic instruction whose target register is longer than the operand, the operand is extended internally to the length of the target register:

- arithmetic operands are sign-extended
- logical operands are extended with zeros.

Examples of arithmetic instructions doing sign extension are AH, AGH, CGFR, and SGFR; examples of logical instructions that extend with zeros are ALGF, CLGF, and CLGFR.

In this section we examined some frequently-used instructions for addition, subtraction, and comparison; they are summarized in Table 77 on page 236.

Function	Operand1	4 bytes		8 bytes	
	Operand2	2 bytes	4 bytes	4 bytes	8 bytes
Arithmetic Add and Subtract (from memory)		AH SH	A S	AGF SGF	AG SG
Arithmetic Add and Subtract (from register)			AR SR	AGFR SGFR	AGR SGR
Logical Add and Subtract (from memory)			AL SL ALC SLB	ALGF SLGF	ALG SLG ALCG SLBG
Logical Add and Subtract (from register)			ALR SLR ALCR SLBR	ALGFR SLGFR	ALGR SLGR ALCGR SLBGR
Arithmetic Compare (to memory)		CH	C	CGF	CG
Arithmetic Compare (to register)			CR	CGFR	CGR
Logical Compare (to memory)			CL CLM	CLGF CLMH	CLG
Logical Compare (to register)			CLR	CLGFR	CLGR

Table 77. Summary of instructions discussed in this section

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
A	5A
AG	E308
AGF	E318
AGFR	B918
AGR	B908
AH	4A
AL	5E
ALC	E398
ALCG	E388
ALCGR	B988
ALCR	B998
ALG	E30A
ALGF	E31A
ALGFR	B91A
ALGR	B90A
ALR	1E
AR	1A

Mnemonic	Opcode
C	59
CG	E320
CGF	E330
CGFR	B930
CGR	B920
CH	49
CL	55
CLG	E321
CLGF	E331
CLGFR	B931
CLGR	B921
CLM	BD
CLMH	EB20
CLR	15
CR	19
S	5B
SG	E309

Mnemonic	Opcode
SGF	E319
SGFR	B919
SGR	B909
SH	4B
SL	5F
SLB	E399
SLBG	E389
SLBGR	B989
SLBR	B999
SLG	E30B
SLGF	E31B
SLGFR	B91B
SLGR	B90B
SLR	1F
SR	1B

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
15	CLR
19	CR
1A	AR
1B	SR
1E	ALR
1F	SLR
49	CH
4A	AH
4B	SH
55	CL
59	C
5A	A
5B	S
5E	AL
5F	SL
B908	AGR
B909	SGR

Opcode	Mnemonic
B90A	ALGR
B90B	SLGR
B918	AGFR
B919	SGFR
B91A	ALGFR
B91B	SLGFR
B920	CGR
B921	CLGR
B930	CGFR
B931	CLGFR
B988	ALCGR
B989	SLBGR
B998	ALCR
B999	SLBR
BD	CLM
E308	AG
E309	SG

Opcode	Mnemonic
E30A	ALG
E30B	SLG
E318	AGF
E319	SGF
E31A	ALGF
E31B	SLGF
E320	CG
E321	CLG
E330	CGF
E331	CLGF
E388	ALCG
E389	SLBG
E398	ALC
E399	SLB
EB20	CLMH

Terms and Definitions

addend
: see *augend*

augend
: When two numbers are added, the number being **aug**mented (the first operand) is the **aug**end, to which the **add**end (the second operand) is **add**ed.

logical arithmetic
: Binary arithmetic and comparison operations with unsigned operands.

minuend
: see *subtrahend*

subtrahend
: When one number is subtracted from another, the number being di**min**ished (the first operand) is the **min**uend, and the number being **sub**tracted (the second operand) is the **sub**trahend.

Programming Problems

Problem 16.1. Write a program that computes three word quantities X, Y, and Z that occupy successive words in memory. Also define a 12-byte character string to occupy the same storage. Compute the contents of the three words as follows:

```
c(X) = B'100000000000000' + X'C7A98' - 231471192,
c(Y) = X'C0FFEE' - C'@#$' - 694895668, and
c(Z) = 1073741823 + X'F194F6' + X'ABCD'.
```

Treat all the quantities as *words* whose values are self-defining terms. A hint: this means that the simplest way to create them is as A-type constants.

Print the hexadecimal and character forms of the 12-byte result (using the PRINTOUT macro, for example).

Problem 16.2. Write a program that computes four values stored in successive words at W, X, Y, and Z. The values are to be computed according to the relations

```
c(W) = c(WA) + c(WB) - 929065920, where
         c(WA) = B'100000000000000' and
         c(WB) = X'1230000'.
c(X) = c(XA) + 50344169 + c(XB), where
         c(XA) = X'5CF17' and
         c(XB) = C'000'.
c(Y) = c(YA) + c(YB) + c(YC), where
         c(YA) = B'11111111',
         c(YB) = X'1261F02', and
         c(YC) = C'ABCD'.
c(Z) = c(ZA) + c(ZB) - c(ZC), where
         c(ZA) = X'CAF75A',
         c(ZB) = B'1000011', and
         c(ZC) = 511686493.
```

All the quantities used in the calculations are four-byte word-aligned constants in memory. Define symbols having length attribute 16 and types C and X to name the same 16 bytes of memory. Calculate W, X, Y, and Z, and print the results of your calculation in character and hexadecimal form (using the PRINTOUT macro, for example).

Problem 16.3. Do as in Problem 16.2, but the four quantities W, X, Y, and Z are defined this time by

```
c(W) = c(WA) + c(WB) - 759375551, where
         c(WA) = B'100000000000000',
         c(WB) = X'CBA98'.
c(X) = c(XA) - c(XB) + 1386388536, where
         c(XA) = X'C0FFEE',
         c(XB) = C'@#$'.
c(Y) = c(YA) + c(YB) + c(YC), where
         c(YA) = B'11111111',
         c(YB) = X'1F7C05',
         c(YC) = C'ABCD'.
c(Z) = c(ZA) + c(ZB) - 975583924, where
         c(ZA) = X'FFFF',
         c(ZB) = -65536.
```

As before, print the 16 bytes of the result as a character string and as a string of 32 hexadecimal digits.

Problem 16.4. Consider the sequence of integers starting

```
0, 1, 2, 3, 6, 11, 20, 37, ...,
```

where (after starting with 0, 1, and 2) each successive term is generated by adding the previous three terms together.

Write a program that will compute and print the first 25 terms of this sequence. (A hint: an appropriate choice of starting values will make it unnecessary to take special actions to print the first few terms.)

Problem 16.5. Suppose you are given three integers A, B, and C, and you are told that they are three successive terms in a sequence. Each term of the sequence was generated by adding the previous three terms together.

Write a program that will generate the *previous* 25 terms of the sequence, for various values of A, B, and C. As a check, you might start with values you found in solving Problem 16.4.

Problem 16.6. Write a program to do the calculations in Figures 92 through 96 for various values of the operands. Use the PRINTOUT macro to display the values of the 64-bit results. For example,

```
         PRINTOUT 17,18
```

displays c(GG1) and c(GG2) in both hex and decimal.

Problem 16.7.(2)+ The Fibonacci[108] series is defined by the relation

```
         F(n+1) = F(n) + F(n-1)          with F(0)=0 and F(1)=1
```

Write a program to calculate and display the numbers in the Fibonacci series starting with F(1) up to the largest value that does not exceed one million.

Problem 16.8.(2)+ Do the same as in Problem 16.7, but now calculate and display the Fibonacci series up to the largest positive value representable in a signed 32-bit binary fullword.

Problem 16.9.(3)+ Do the same as in Problem 16.8, but format and print the results using the CONVERTO and PRINTLIN macros.

Problem 16.10.(3) Calculate the numbers in the Fibonacci series (described in Problem 16.7) up to the maximum positive value representable using 64-bit binary arithmetic, and format and print the results using the CONVERTO and PRINTLIN macros.

Problem 16.11.(2)+ Assemble the following program:

---

[108] Named after Leonardo of Pisa, known as Fibonacci.

```
P16_11   CSect ,
         Using *,12
         LR    12,15
         A     15,X
         BASR  12,15
X        DC    F'18'
         DC    F'4'
Exit     BR    14
         L     10,X-4
         B     X-4(10)
         End   P16_11
```

Study the object code carefully, and explain what each instruction does and how it does it.

Problem 16.12.(2)+ Write and execute a program to test the results of Exercise 16.2.16 above. (Remember that the PRINTOUT macro will display both register contents and CC settings.)

17. Binary Shifting

```
    11      777777777777
   111      777777777777
  1111      77        77
    11               77
    11              77
    11              77
    11             77
    11             77
    11            77
    11            77
1111111111       77
1111111111       77
```

The multiplication and division instructions in Section 18 are often combined with shift operations, so we'll start with instructions that shift data within a single general register or pair of general registers.

The general register shift instructions are summarized in Table 78. Nine operate on data in 32-bit registers, and five operate on 64-bit registers. The notation "(32+32)" means that 64 bits are shifted in an even-odd pair of 32-bit general registers. There are no double-length shifts of 128-bit operands "(64+64)" in an even-odd pair of 64-bit general registers.[109]

We say that single-length shifts operate on bits in a single 32- or 64- bit register, and double-length shifts operate on bits in an even-odd pair of registers.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
88	SRL	RS	Shift Right Logical (32)	89	SLL	RS	Shift Left Logical (32)
8A	SRA	RS	Shift Right Arithmetic (32)	8B	SLA	RS	Shift Left Arithmetic (32)
8C	SRDL	RS	Shift Right Double Logical (32+32)	8D	SLDL	RS	Shift Left Double Logical (32+32)
8E	SRDA	RS	Shift Right Double Arithmetic (32+32)	8F	SLDA	RS	Shift Left Double Arithmetic (32+32)
EB0C	SRLG	RSY	Shift Right Logical (64)	EB0D	SLLG	RSY	Shift Left Logical (64)
EB0A	SRAG	RSY	Shift Right Arithmetic (64)	EB0B	SLAG	RSY	Shift Left Arithmetic (64)
EB1C	RLLG	RSY	Rotate Left Logical (64)	EB1D	RLL	RSY	Rotate Left Logical (32)

Table 78. General register shift instructions

These RS-type instructions differ from other RS-type instructions: the shaded portion of the instruction (where the R_3 register specification digit would be) in Table 79 on page 243 is *ignored* when the instructions are executed.

[109] At the time of this writing. But new instructions are added regularly to the System z architecture, so check the *Principles of Operation*.

opcode	R_1		B_2	D_2

Table 79. RS-type shift instruction

Thus, the Assembler makes no provision for specifying a value in that field, and sets it to zero. The operand field entry for shift instructions is written in either of the two forms

```
R1,D2(B2)   (explicit address)
R1,S2       (implied address)
```

and *no* R_3 operand is specified.

The RSY-type shift instructions *do* have an R_3 operand, as shown in Table 80. For these instructions, the source operand is in the R_3 register and the result goes into the R_1 register. We'll see examples using the R_3 operand when we discuss these instructions.

opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

Table 80. RSY-type instruction format

When executed, none of the logical shift instructions change the CC setting, while all of the arithmetic shifts treat the shifted data as signed, and set the CC to indicate the status of the result.

For all shift instructions, the number of bit positions to be shifted is determined from the low-order six bits of the Effective Address; this allows *actual* shift amounts only between 0 and 63. That is, the shift count is the remainder obtained when the Effective Address is divided by 64:

```
shift count = Effective Address (modulo 64).
```

This means, for example, that a shift amount specified by an Effective Address of 66 actually shifts only 2 positions when executed.

Shift Amounts

Shift instructions can specify at most 63 shifts.

First, we'll describe the *unit shift*, and then look at the eight RS-type instructions, all of which involve 32-bit registers.

17.1. Unit Shifts

To illustrate the behavior of various shift instructions, we'll assume that the source register starts with the contents illustrated in Figure 104.

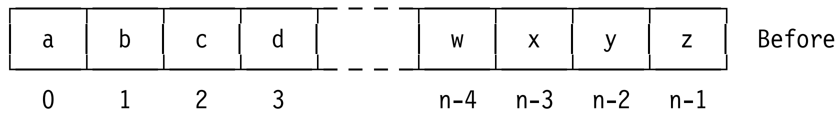


Figure 104. Register contents before shifting

The bit positions are numbered from 0 to n−1, where n is the number of bits participating in the shift.

The basic shift operation is the *unit shift,* in which each bit moves right or left by one bit position. The digit position at the *right* (low-order) end of the register behaves identically for logical and arithmetic left and right shifts, but the bit at the *left* (high-order) end of the register is treated differently.

For logical shifts, the vacated bit position at either end of a register is always set to zero, and the bit shifted off the opposite end is lost and ignored. This is illustrated in Figures 105 and 106.

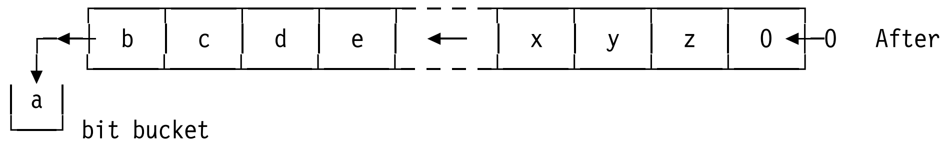


Figure 105. Logical unit shift left

The "bit bucket" doesn't really exist; it just means that the lost bit vanishes.[110]

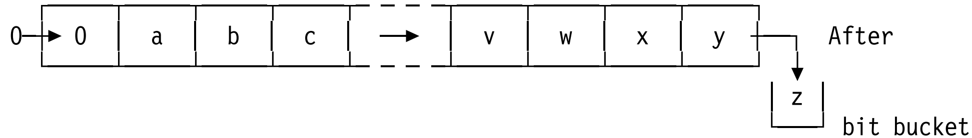


Figure 106. Logical unit shift right

For arithmetic right shifts, the rightmost bit is lost and ignored, and the sign bit is *duplicated* to preserve the arithmetic integrity of the operand. This is illustrated in Figure 107.

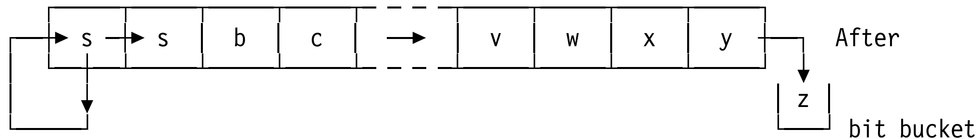


Figure 107. Arithmetic unit shift right

For arithmetic left shifts, the vacated bit position at the right end is set to zero, and the sign bit is *not* shifted; it doesn't move. However, the bit immediately to the right of the sign bit is lost. This is illustrated in Figure 108.

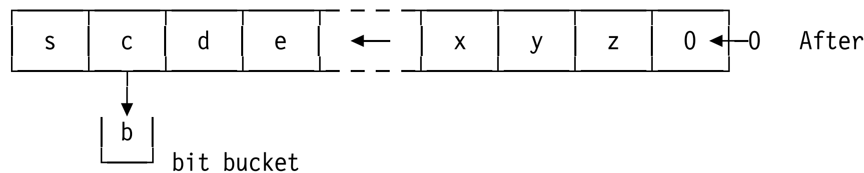


Figure 108. Arithmetic unit shift left

Again, the sign of the operand is preserved. Because arithmetic left shifts may lose a significant bit, an overflow condition can occur; we'll see how this happens when we look at the arithmetic shift instructions in Section 17.4.

To illustrate unit shifts, suppose c(GR8) is `X'87654321'`, or

```
1000 0111 0110 0101 0100 0011 0010 0001
```

in binary, and a unit logical left shift in GR8 is executed. Each of the bits moves one position to the left, and the result in GR8 will be

```
0000 1110 1100 1010 1000 0110 0100 0010
```

in binary, or `X'0ECA8642'`. The leftmost one-bit was lost, and a zero-bit was introduced at the right. Similarly, if we again start with `X'87654321'` and execute a unit logical right shift in GR8, each bit moves one position to the right, and the result will be

```
0100 0011 1011 0010 1010 0001 1001 0000
```

[110] When I took my first programming class, we were all taken to see the computer; its operation was slowed so we could watch it shift, add, etc. After showing the shifts the instructor paused, because a student always asked "What happens to the bits shifted off the end?" An engineer would then open a door on the end of the machine and hold up a small silver bucket, saying gravely that the bits had to be emptied after every 8 hours of operation. Some of us never realized it was a joke.

in binary, or X'43B2A190'.

The execution of a shift instruction is simple: it simply performs the number of unit shifts specified by the low-order 6 bits of its Effective Address.

Exercises

17.1.1.(1) What shift amounts are represented by each of the following Effective Addresses?

1. X'EDCBA987'
2. X'12345678'
3. X'87654321'
4. X'00000FED'
5. X'FFFFFFFF'
6. X'27A49FC1'
7. X'6789ABC0'

17.2. Single-Length Logical Shifts

The simplest shifting instructions are SRL (Shift Right Logical) and SLL (Shift Left Logical). In most of the following examples, bit patterns will be represented in hexadecimal.

To perform a unit logical left shift of the contents of R8, we can execute the instruction

```
         SLL   8,1(0)               Shift GR8 left 1 bit position
```

Suppose GR8 again contains X'87654321' and GR3 contains X'82F3A2B5', executing the logical right-shift instruction

```
         SRL   8,16(3)
```

first causes the Effective Address to be computed as

```
X'82F3A2B5' + X'010' = X'82F3A2C5'
```

of which the rightmost six bits are B'000101'. Thus it shifts right five bit positions, leaving

```
0000 0100 0011 1011 0010 1010 0001 1001
```

in binary, or X'043B2A19' as the result in GR8.

In these examples, we saw that the original contents of GR8 were not preserved: that is, the shifts can be thought of as "destructive". All the RS-type shifts use the same register (or register pair) as the source and target of the operation. The RSY-type shifts let you preserve the original source operand if you like.

The SLL instruction is the most commonly used logical shift. It is often used to multiply index values by a power of two (such as the length of an operand in memory) prior to executing an RX-type instruction for which the shifted register is the index register. We will see many such uses in discussing looping and indexing in Section 22.

- Suppose the word at **Index** contains a small positive integer N that is to be used to index into a table of words starting at the word named **Tab**. To load the N-th of those words into GR0, we could write a sequence of instructions like the following:

  ```
           L    1,Index              Get index word
           SLL  1,2                  Shift left 2 bits (multiply by 4)
           L    0,Tab-4(1)           Load N-th word into GR0
  ```

  The shift left by two bit positions is needed so that we access the N-th *word* (not the N-th *byte*) in the table; and we must address the table at **Tab-4** because if the integer at **Index** is 1, we should access the first word at Tab. If N is 1, indexing will add 4 to **Tab-4**, giving the address of **Tab** as desired.

- Suppose we want to set the leftmost seven bits of register 8 to zero, leaving the other bits unchanged. Then we could execute the two instructions

```
         SLL   8,7                 Shift left 7 places, drop off bits
         SRL   8,7                 Shift right 7 places, bring in zeros
```

and the leftmost 7 bits are replaced by zeros.

- As another example, suppose we need to align the address in GR6 to a doubleword boundary. That is, we will force the value in GR6 to be a multiple of 8 in such a way that if it is not already so, the next higher multiple of 8 will be chosen.

  This can be done very simply:

```
         AL   6,=F'7'              Force carry if possible
         SRL  6,3                  Drop off three bits
         SLL  6,3                  Multiply by 8
```

Figure 109. Rounding an integer to the next higher multiple of 8

  The presence of any 1-bit in the three rightmost bits of the original number in GR6 will cause a carry into the 2^3 bit position (bit number 28 of GR6).

- Suppose we have a large table of six-byte data items containing a mix of integer and character data. Each table entry is aligned on a halfword boundary. Suppose also that the data is arranged so that the first three bytes contain a signed 24-bit two's complement integer, and the remaining three bytes contain the character data (see Figure 110).

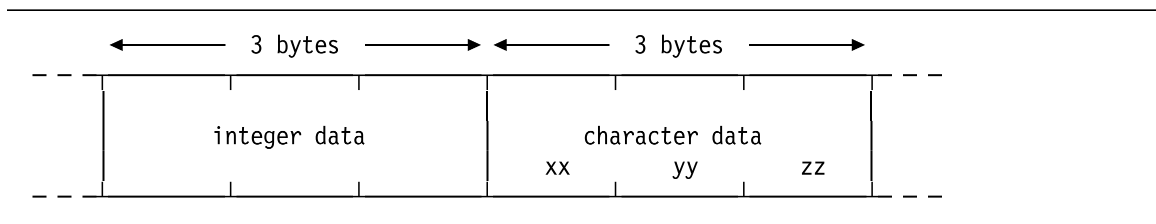


Figure 110. A 6-byte data entry

  Space for typical table entry might have been reserved with DS statements such as

```
Entry    DS   0XL6                 Define name of 6-byte data entry
IntPart  DS   FL3                  Give name to integer part
CharPart DS   CL3                  And to the character part
```

Figure 111. Storage definitions for a 6-byte data entry

  We want to retrieve the integer value from a data entry and place it into GR5 where it will be used for some purpose in the program, and then store it from GR5 back into memory in the format illustrated in Figure 110. We can see that L and ST instructions cannot be used, because the operands are neither 4 bytes long nor correctly aligned in memory; similarly, LH and STH handle only two of the three bytes.

  Now, suppose GR12 contains the address of the first byte of a data entry. The instructions needed to load the integer value into GR5 are shown in Figure 112. (Assume for the moment that the data entry contains X'F01234xxyyzz'; for now, we'll ignore the three characters represented by “xxyyzz”.)

```
      LH    5,0(0,12)     c(GR5) = X'FFFFF012', leftmost 16 bits
      SLL   5,8           c(GR5) = X'FFF01200', move left 8 bits
      IC    5,2(0,12)     c(GR5) = X'FFF01234', insert last 8 bits
      - - -               - - do some calculations with the value
      STC   5,2(,12)      Store rightmost 8 bits
      SRL   5,8           Position remaining 16 Bits
      STH   5,0(0,12)     Store high-order part
```

Figure 112. Using shift instructions for a 6-byte data item

  The arrangement of data in memory usually depends on the requirements of the application, as well as on considerations of ease of programming or speed of execution.

This example might tempt you to manipulate *characters* by inserting and shifting them in the general registers. Resist that that temptation until after we have examined instructions designed specifically for managing character data in Section 25.

17.2.1. Three-Operand Shift Instructions

SRLG and SLLG are the 64-bit equivalents of SRL and SLL. They behave exactly as the 32-bit shifts, with a useful extension: rather than specifying only a single operand register (as in Table 79 on page 243), these two RSY-type instructions specify separate source and target registers. The source operand is taken from GG R_3, shifted by the specified amount, and placed into the target operand, GG R_1. The operand field is is written

```
R1,R3,D2(B2)   (explicit address)
R1,R3,S2       (implied address)
```

Table 80 on page 243 shows the format of an RSY-type instruction.

If you specify different register numbers for R_3 and R_1, the shift is “nondestructive” because the source operand in GG R_3 is unchanged. If you specify the same register number for both R_3 and R_1, the shift is “destructive”, just like the shifts in 32-bit general registers.

To illustrate, consider these instructions:

```
L     0,=A(X'12345678')  c(GR0) initialized
SLL   0,9                c(GR0) = X'68ACF000'
```

and the contents of GR0 is changed. For these instructions:

```
LG    1,=XL8'123456789ABCDEF0'    c(GG1) initialized
SLLG  0,1,9              c(GG0) = X'68ACF135 79BDE000'

LG    1,=XL8'123456789ABCDEF0'    c(GG1) initialized
SRLG  0,1,9              c(GG0) = X'00091A2B 3C4D5E6F'
```

in both cases, the contents of the GG1 source register is unchanged. Otherwise, the instructions for shifting operands in 64-bit registers behave the same way as their equivalents for 32-bit registers.

Exercises

17.2.1.(2)+ Suppose the string of bytes beginning at **BStrg** is to be considered as a string of *bits*. Given an integer K stored in the word at **KK**, write a code sequence to place in GR0 the value of bit K of the string. (Remember to start numbering the bits at zero.)

17.2.2.(2) A word integer at **K** has value between 0 and 7. Write a code sequence using shifts that will store at **KthBit** a byte containing a single 1-bit at a position determined by the integer at **K**. That is, if c(K)=6, then c(KthBit) = X'02'. (Remember that bits in a byte are numbered from 0 to 7!)

17.2.3.(2) Rewrite Exercise 17.2.2 to use no shifts, but define an appropriate 8-byte table. Which code sequence is shorter? Simpler?

17.2.4.(1)+ The SLL instruction shifts data in a 32-bit general register. How many bit positions will be shifted if you specify

```
SLL   0,33   ?
```

17.2.5.(2) The word at DPG contains 4 bytes; write instructions to put those four bytes into GR1 in reverse order. Thus, if c(DPG) is X'12345678', c(GR1) will be X'78563412'.

17.2.6.(2)+ GR0 contains a positive, nonzero number. Write a set of instructions that will shift the number to the left until there is a 1-bit in bit position 1 of GR0 (the bit immediately to the right of the sign bit). In GR1, put the number of positions shifted. Remember that the number in GR0 must be positive when the instruction sequence terminates.

17.2.7.(2)+ Given these two constants at X and Y:

```
X        DC    FL3'1234567'
Y        DC    FL3'7654321'
```

Write instructions to add the two numbers and store their sum as a 24-bit number at W. If the sum overflows and cannot be represented correctly, branch to OverFlo.

What are the hexadecimal representations of the constants at X and Y? What is the representation of the result stored at W, and does the sum overflow?

17.2.8.(2) What will be the result of executing this instruction?

```
         SLL   n,n(n)
```

17.2.9.(2)+ In the example following Figure 109 on page 246, does it matter if the 3-byte integer data is signed or unsigned? Explain.

17.3. Double-Length Logical Shifts

The double-length logical shift instructions SLDL (Shift Left Double Logical) and SRDL (Shift Right Double Logical) work in exactly the same way as SLL and SRL, except they shift the 64 bits in a *pair* of even-odd 32-bit registers. The register specified by the first operand (R_1) must be an even-numbered register; otherwise a specification exception will occur. The next higher-numbered register is the low-order half of the double-length register pair. Bits right-shifted out of the right end of GR R_1 enter the left end of GR R_1+1, and vice versa for left shifts. (Figure 10 on page 46 shows paired general registers.)

Revisiting the example in Figure 109 on page 246, here is another way to round an integer to the next higher multiple of 8 if it is not already a multiple of 8.

```
         SR    7,7                Clear GR7 to zero
         SRDL  6,3                Shift three bits into GR7 from GR6
         LTR   7,7                Test whether the bits are zero
         BZ    A                  Branch if yes
         A     6,=F'1'            If not, add 1 to GR6
A        SLL   6,3                Finally, multiply GR6 by 8
```

First, we clear GR7 by subtracting it from itself, a fast and simple way to do this. Then, we use a shift instruction to divide by 8. The double-length shift moves the three “remainder” bits into the three high-order bit positions of GR7. The BZ instruction branches only if the remainder bits are all zero: that is, if the number in GR6 was already a multiple of 8. If any remainder bit is nonzero, 1 is added to GR6. Finally, GR6 is shifted left 3 bit positions to give the correct multiple of 8.

As another example, suppose a positive nonzero integer word at **N** is to be shifted right as many places as necessary to ensure that its rightmost bit is nonzero. Here are two ways we might do this:

1. Shift left from GR5 into GR4, until only zero-bits remain in GR5. That is, if two right shifts of the integer at N were actually needed, we will do 30 double-length left shifts.

```
         L     5,N                Get integer from N
         L     4,=F'0'            Clear GR4
ShiftL   SLDL  4,1                Shift left one bit position
         LTR   5,5                Test remaining bits in GR5
         BNZ   ShiftL             Repeat if not zero
         ST    4,N                Store result
```

Figure 113. Shifting to make the low-order bit one (1)

2. This time, we shift right, testing “lost” bits:

```
         L     4,N                Get integer from N
ShiftR   SRDL  4,1                Shift right once
         LTR   5,5                Test sign bit of GR5
         BNM   ShiftR             Branch if not minus
         SLDL  4,1                Move the bit back
         ST    4,N                Store result
```

Figure 114. Shifting to make the low-order bit one (2)

This second example will also work for negative integers if arithmetic shift instructions are used.

These examples illustrate simple *loops*, instructions that are repeated as many times as necessary to obtain a desired result or condition. Loops are an important aspect of programming; special System z branch instructions simplify coding of loops.[111]

Suppose that in a certain application we need to store some integer data in a very compact format. The integer values are unsigned and are small enough that we can squeeze four integers into a 32-bit word as shown in Figure 115. (Section 17.6 will describe *how* you can define these four values in a word.)

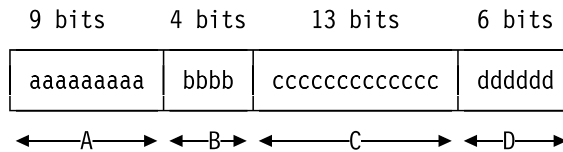


Figure 115. Four integers packed in a 32-bit word

Suppose the four packed integers are stored at **DataWord** and we want to extract the second integer (the four bbbb bits) and store their value in the word at **BVal**. We can do this with the instructions in Figure 116:

```
         L     0,DataWord     Load 32 bits
         SLL   0,9            c(GR0)=bbbbcccccccccccccdddddd000000000
         SRL   0,28           c(GR0)=0000000000000000000000000000bbbb
         ST    0,BVal         Store value of b-bits
```

Figure 116. Extracting one packed integer from a 32-bit word

The SLL instruction shifts all the a bits off the left end of GR0, and the SRL instruction shifts all but the four b bits off the right end of GR0, leaving only the four bbbb bits right-adjusted in GR0.

To illustrate a more general technique, we will write instructions that extract the integers from their compacted word format in a memory area named **DataWord**, separating them into individual words named **First**, **Second**, **Third**, and **Fourth**. In Figure 117 on page 250, the comment statements show the binary contents of registers GR0 and GR1; the integers to be unpacked are named A, B, C, and D as shown in Figure 115. In Figure 117 on page 250, a letter “x” represents a bit whose value is unknown, and 0 is a zero bit. We will shift each integer from the right end of GR0 into GR1, where it will be right-justified in GR1 and stored. This example uses only right shifts.

As mentioned in Section 13.3 on page 162, the EQU instruction assigns the value of the operand to the name-field symbol. This symbolic technique is very useful if the sizes of the fields must be changed, because the shift instruction operands will be adjusted automatically by the Assembler.

[111] These special “Branch on Index” and “Branch on Count” instructions neither examine nor change the CC. We will investigate them in Section 22.

```
LA       EQU   9                 Define bit length of integer A
LB       EQU   4                 Length of B
LC       EQU   13                Length of C
LD       EQU   6                 Length of D
         L     0,DataWord        Load data fullword into GR0
* c(GR0) = B'aaaaaaaaabbbbccccccccccccccdddddd'
* c(GR1) = B'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
         SRDL  0,LD              Shift 6 bits in both registers
* c(GR0) = B'000000aaaaaaaaabbbbccccccccccccc'
* c(GR1) = B'ddddddxxxxxxxxxxxxxxxxxxxxxxxxxx'
         SRL   1,32-LD           Move D to right end of GR1
* c(GR1) = B'00000000000000000000000000dddddd'
         ST    1,Fourth          Store fourth integer D at FOURTH
         SRDL  0,LC              Shift 13 bits in both registers
* c(GR0) = B'0000000000000000000aaaaaaaaabbbb'
* c(GR1) = B'ccccccccccccc0000000000000000000'
         SRL   1,32-LC           Move C to right end of GR1
* c(GR1) = B'0000000000000000000ccccccccccccc'
         ST    1,Third           Store third integer C at THIRD
         SRDL  0,LB              Shift 4 bits in both registers
* c(GR0) = B'00000000000000000000000aaaaaaaaa'
* c(GR1) = B'bbbb0000000000000000000ccccccccc'
         ST    0,First           Store first integer A from GR0
         SRL   1,32-LB           Position second integer B in GR1
* c(GR1) = B'0000000000000000000000000000bbbb'
         ST    1,Second          Store second integer B at Second
```

Figure 117. Unpacking four unsigned integers using right shifts

We can also shift the integers left, from the left end of GR1 into the right end of GR0, but we must clear GR0 each time before shifting.

```
         SR    2,2               Constant zero for clearing GR0
         L     1,DataWord        Get data fullword in GR1
         LR    0,2               Clear GR0
         SLDL  0,LA              Shift 9 bits into GR0 from GR1
         ST    0,First           Store first integer
         LR    0,2               Clear GR0
         SLDL  0,LB              Shift 4 bits into GR0
         ST    0,Second          Store second integer
         LR    0,2               Clear GR0
         SLDL  0,LC              shift 13 bits into GR1 into GR0
         ST    0,Third           store third integer
         SRL   1,LA+LB+LC        Reposition fourth integer in GR1
         ST    1,Fourth          Store fourth integer
```

Figure 118. Unpacking four unsigned integers using left shifts

We have used LR instructions to clear GR0, rather than subtracting it from itself. Similarly, in this example the final "`SRL 1,LA+LB+LC`" shift replaces the LR and SLDL used in the first three steps, because it results in less code and faster execution. The overall saving is quite small, but this illustrates a small economy that could result in significant savings if this sequence is frequently executed.

Exercises

17.3.1.(2)+ Suppose your CPU has only single-length logical shift instructions (SLL, SRL). A word at **DataWord** is to be shifted logically to the left, as though it was the *high-order* word of a pair of general registers. Write an instruction sequence that simulates a double-length left shift of N bit positions, where N is a halfword integer at **NShifts**. Assume $0 \leq N < 32$, and that the simulated low-order "register" contains zero. Store the result at a doubleword named **DWord**.

17.3.2.(2) Do the same as in Exercise 17.3.1, and again assume you must do a double-length logical left shift of a 32-bit word in the high-order half. This time, assume $0 \le N < 64$.

17.3.3.(2)+ Do the same as in Exercise 17.3.1, but simulate a double-length logical *right* shift of N places, where $0 \le N < 32$, again assuming that the low-order half of the original 64-bit operand is zero.

17.3.4.(2) Do the same as in Exercise 17.3.3, but now assume $0 \le N < 64$.

17.3.5.(3)+ Do the same as in Exercise 17.3.1, but now assume the initial data is in a *doubleword* at **DWData**, and store the *left-shifted* result at **DWord**.

17.3.6.(3)+ Do the same as in Exercise 17.3.5, but now assume the initial data is in a doubleword at **DWData**, and store the *right-shifted* result at **DWord**. (Remember that $0 \le N < 32$.)

17.3.7.(2) Do the same as in Exercise 17.3.5, but now assume $0 \le N < 64$.

17.3.8.(2) Do the same as in Exercise 17.3.6, but now assume $0 \le N < 64$.

17.3.9.(3) There is a word at **OLD** into which four positive integers have been packed as illustrated in Figure 115 on page 249. Write a code sequence to rearrange the four unsigned integers into a new word format, in which the first integer occupies the first seven bits, the second integer occupies the next two, the third is expanded to occupy the next fifteen bits, and the fourth integer occupies the last eight bits. Store the result at **NEW**.

17.3.10.(3)+ Suppose four unsigned integers are stored in the words named **FIRST**, **SECOND**, **THIRD**, and **FOURTH**. Write a code sequence that will pack the integers from those words into a word at **NEW** in the format illustrated in Figure 115 on page 249.

17.3.11.(3) As in Exercise 17.3.10, assume we wish to pack the four integers at **FIRST**, **SECOND**, **THIRD**, and **FOURTH** into a word at **NEW**. The number of bits to be allocated to each integer in its packed form is given as the value of the four positive halfword integers stored at **L1**, **L2**, **L3**, and **L4** respectively. We know that

```
c(L1) + c(L2) + c(L3) + c(L4) = 32.
```

The integers to be packed are stored in the logical representation.

17.3.12.(3) Rewrite Exercise 17.3.11 assuming that the values are to be stored in the arithmetic representation.

17.3.13.(2) What will happen in Figures 113 and 114 if c(N)=0?

17.3.14.(3)+ A common mathematical notation is the "ceiling" function. If a number x has integer part p and fraction part q, we write "p.q" to represent x. The "ceiling" function is defined as:

```
if q = 0, Ceiling(x) = p,
if q > 0, Ceiling(x) = p+1.
```

Suppose there is a nonnegative integer N stored in the word at **NN**. Write a code sequence that will leave Ceiling(N+N/2) in GR1.

17.3.15.(4)+ Rewrite Exercise 17.3.10 to repack the four unsigned integers into a new word, but include tests to check that the values will fit into the fields provided for them in the packed word. To indicate whether or not each of the integers fits into its allotted field, set the bytes at **FLAG1**, **FLAG2**, **FLAG3**, and **FLAG4** zero if the value will fit, and to nonzero if the value will not fit.

17.3.16.(2) GR0 contains a 32-bit number considered as a bit pattern. Write a code sequence that will place the same bit pattern into GR1, but reversed from right to left within the register.

17.3.17.(2)+ The word at **Data** contains information to be shifted *circularly*: that is, bits shifted off one end of the register should reappear at the other end. For example, a circular left shift of the operand `X'12345678'` by 12 bit positions would produce `X'45678123'`. Write instructions

(not using RLL!) to shift c(Data) circularly to the left by N places, where N is a nonnegative word integer stored at **NShifts**. Can you do this using only single-length shifts?

17.3.18.(2)+ Modify the coding of Exercise 17.3.17 so that if N is negative, the shift is a circular *right* shift instead.

17.3.19.(3)+ A programmer wanted to display the hex digits in a byte string starting at Hex as a string of EBCDIC characters starting at Chars, with each EBCDIC character representing a single hexadecimal digit. The length of the byte string with the hex digits is stored as a halfword binary integer stored at Len. He wrote:

```
         LH    0,Len             Get length of source string in GR0
         L     2,=A(Hex)         Addr of start of hex string in GR2
         L     3,=A(Chars)       Addr of start of char string in GR3
GetAByte SR    4,4               Clear GR4 for a work register
           IC    4,0(,2)         Get a byte from hex string
           SRDL  4,4             Move high-order hex digit in GR4
           SRL   5,28            And low-order hex digit in GR5
           IC    4,EBCDIC(4)     Get character form of high digit
           IC    5,EBCDIC(5)     Get character form of low digit
           SLL   4,8             Make room in GR4 for second byte
           ALR   4,5             Now have both characters in GR4
           STCM  4,B'0011',0(3)  Store both chars in output string
           AH    2,=H'1'         Increment input pointer
           AH    3,=H'2'         Increment output pointer
           SH    0,=H'1'         Reduce input byte count by 1
         BP    GetAByte          If count > 0, do another byte
         - - -
EBCDIC   DC    C'0123456789ABCDEF'    EBCDIC form of hex digits
```

Does this work? Explain.

17.4. Arithmetic Shift Instructions

The arithmetic shift instructions are similar to the logical shift instructions, except for the setting of the CC and the treatment of the sign bit. The instructions are SLA (Shift Left Arithmetic), SRA (Shift Right Arithmetic), SLDA (Shift Left Double Arithmetic), and SRDA (Shift Right Double Arithmetic). The CC settings after arithmetic shift instructions are similar to those for the arithmetic add and subtract instructions:

Operation	CC Setting and Meaning
Left shift	0: Result is zero 1: Result is < zero 2: Result is > zero 3: Result has overflowed
Right shift	0: Result is zero 1: Result is < zero 2: Result is > zero 3: Cannot occur

Table 81. CC settings for arithmetic shift instructions

As we saw in Figure 107 on page 244, for *right* shifts the sign bit is duplicated (or *extended*) in the vacated sign position after each unit shift, to preserve the arithmetic integrity of the shifted operand.

To illustrate the difference between logical and arithmetic shifts, suppose a right shift of two bits is performed on a register containing X'FFFFFFF8':

```
         L     0,=F'-8'                 L     0,=F'-8'
         SRL   0,2                      SRA   0,2
```

After the SRL logical shift, c(GR0)=X'3FFFFFFE', because two zero bits were inserted at the left; after the SRA arithmetic shift, c(GR0)=X'FFFFFFFE', because the sign bit has been duplicated. For positive operands, the SRL and SRA instructions will leave identical results in the register; SRA will set the CC as shown in Table 81 on page 252, but SRL will leave the CC unchanged. The SRDA instruction is similar to SRA, except that an even-odd register pair is shifted as a single 64-bit entity.

A typical use of SRDA is to create a correctly-signed 64-bit dividend for a fixed-point divide instruction, as we will see in Section 18:

```
         L     0,Dividend          32-bit number in GR0
         SRDA  0,32                Sign-extend to 64-bit length in (GR0,GR1)
         D     0,Divisor           Divide by 32-bit number
```

The sign bit of the word at **Dividend** has been extended by the SRDA instruction to fill GR0.

For arithmetic *left* shifts, the situation is a little more complicated, as we saw in Figure 108 on page 244. When an operand is shifted left one or more significant bits may be lost; though lost, they are not ignored! An arithmetic left shift (1) *always* retains the original sign bit, and (2) indicates an overflow if any bit shifted out of the position just to the right of the sign is different from the sign bit. This is a fixed-point overflow, and may cause a program interruption with the Interruption Code set to 8.

The following instructions will produce the results indicated in the remarks fields:

```
         L     0,=F'-8'           c(GR0)=FFFFFFF8, CC unchanged
         SRL   0,2                c(GR0)=3FFFFFFE, CC unchanged
         SLA   0,4                c(GR0)=7FFFFFE0, CC set to 3 (Overflow)
```

When executing the SLA instruction, one 0-bit and three 1-bits are shifted out of the bit position immediately to the right of the sign bit. Because the sign bit is zero after the SRL instruction, the first one-bit to be shifted out of the bit position just to the right of the sign signals the overflow condition, since it differs from the sign.

We can use the ICM, STCM, and SRA instructions to simplify the example in Figure 112 on page 246:

```
         ICM   5,B'1110',0(12)     c(GR5) = X'F01234??'
         SRA   5,8                 c(GR5) = X'FFF01234'
         - - -                     Compute something
         STCM  5,B'0111',0(12)     Store result back
```

As indicated in Table 81 on page 252, a CC value of 3 is not possible after the SRA and SRDA instructions, because there can be no overflow. For SLDA and SRDA, the result tested is a *double-length* operand, so these instructions provide a simple way to test whether both registers contain zero. Both SRDA 0,0 and SLDA 0,0 will set the CC to zero if the register pair (GR0,GR1) both contain zeros.

An important use of the arithmetic shift operations is to multiply by positive and negative powers of two. Since the bits of an operand shifted left by a unit shift appear with a weight (in the sum forming the value of the operand) that has *increased* by two, so long as no significant bits are lost and no overflow occurs, an arithmetic left shift of n places corresponds to multiplication by 2^n.

Similarly, for a unit right shift, each bit has a weight that has *decreased* by two, so that an arithmetic right shift of n places corresponds to division by 2^n. Because such a “division” might seem to produce fractional results, we must check what happens when bits are lost. Consider these sequences:

```
L     3,=F'5'       c(GR3) = 00000005 = +5
SRA   3,1           c(GR3) = 00000002 = +2 (1-bit lost)

L     3,=F'-5'      c(GR3) = FFFFFFFB = -5
SRA   3,1           c(GR3) = FFFFFFFD = -3 (1-bit lost)
```

As we expect, the lost bit in the first case results in the fractional part of (5/2) being discarded, so the result is simply 2. In the second case the result is −3, not −2; this is because the truncation of the fraction part of a number in the two's complement representation has the effect of always forcing the result to the next *algebraically lower* integer value. (See Exercises 17.4.9 and 17.4.14.)

As a simple example, suppose we wish to truncate the integer in GR9 to the next algebraically lower multiple of 16, unless it is already a multiple of 16. Both of the following achieve the desired result.

```
SRA   9,4                          SRL   9,4
SLA   9,4                          SLL   9,4
```

Either the logical or arithmetic shifts can be used, because whatever bit is shifted out of the sign position by the SRL instruction will be put back by the SLL. If a CC setting is desired to indicate the status of the result, arithmetic shifts must be used.

To conclude our discussion of shifting, we revisit the problem of retrieving the data packed in the word pictured in Figure 115 on page 249, but now assuming that each of the four integers is *signed* rather than logical. The following code segment separates and stores the four signed integers as required; we again use the symbols LA, LB, LC, and LD to represent the bit lengths of the fields, as in Figure 117 on page 250.

```
L     0,DataWord        Get data word into GR0
SRDA  0,LD              Shift 6 bits into GR1
SRA   1,32-LD           Sign-extend to right
ST    1,Fourth          Store fullword result D
SRDA  0,LC              Shift off 13 more bits into GR1
SRA   1,32-LC           Shift with sign extension
ST    1,Third           Store signed result of C
SRDA  0,LB              Shift off next 4 bits for B
SRA   1,32-LB           Sign-extend second integer
ST    1,Second          Store final result of B
ST    0,First           Store correct first integer A
```

Figure 119. Unpacking four signed integers

As noted in Section 17.2, the instructions for shifting operands in 64-bit registers behave just like the equivalent instructions for shifting operands in 32-bit registers. To illustrate, consider these right shift instructions:

```
L     0,=A(X'12345678')  c(GR0) initialized
SRA   0,9                c(GR0) = X'00091A2B'
```

The contents of GR0 is changed. For these instructions,

```
LG    1,=AD(X'123456789ABCDEF0')  c(GG1) initialized
SRAG  0,1,9              c(GG0) = X'00091A2B 3C4D5E6F'
```

the contents of the source register, GG1, is unchanged. If we initialize the source register with a negative number, the sign bit is propagated:

```
L     0,=A(X'87654321')  c(GR0) initialized
SRA   0,9                c(GR0) = X'FFC3B2A1'
```

and the contents of GR0 is changed. For these instructions,

```
LG    1,=XL8'FEDCBA9876543210'    c(GG1) initialized
SRAG  0,1,9              c(GG0) = X'FFFF6E5D 4C3B2A19'
```

GG1 is again unchanged.

Left arithmetic shifts may cause overflow:

```
L     0,=A(X'87654321')  c(GR0) initialized
SLA   0,9                c(GR0) = X'CA864200', CC=3

LG    1,=XL8'FEDCBA9876543210'    c(GG1) initialized
SLAG  0,1,9              c(GG0) = X'B97530EC A8642000', CC=3
```

Double-Length Shifts

The double-length shift instructions (SRDA, SLDA, SRDL, SLDL) always require an even-odd pair of general registers.

Exercises

17.4.1.(2) Suppose your CPU has only single-length arithmetic shift instructions (SLA, SRA). There is a word at **DataWord** that is to be shifted arithmetically to the left, as though it was the high-order word of a pair of general registers. Write an instruction sequence that simulates a double-length arithmetic left shift of N bit positions, where N is a halfword integer at **NShifts**. Assume $0 \leq N < 32$, and that the simulated low-order "register" contains zero. Store the result at a doubleword named **DWord**. If you can, show whether or not the CC setting is correct at the end of your instruction sequence.

17.4.2.(3) Do the same as in Exercise 17.4.1, and again assume you must do a double-length arithmetic left shift of a 32-bit word in the high-order half. This time, assume $0 \leq N < 64$.

17.4.3.(3)+ Do the same as in Exercise 17.4.1, but simulate a double-length arithmetic *right* shift of N places, where $0 \leq N < 32$, and still assuming that the low-order half of the original operand is zero.

17.4.4.(3) Do the same as in Exercise 17.4.3, but now assume $0 \leq N < 64$.

17.4.5.(3)+ Do the same as in Exercise 17.4.1, but now assume the initial data is in a *doubleword* at **DWData**, and store the *left-shifted* result at **DWord**.

17.4.6.(3)+ Do the same as in Exercise 17.4.5, but now assume the initial data is in a doubleword at **DWData**, and store the *right-shifted* result at **DWord**. (Remember that $0 \leq N < 32$.)

17.4.7.(3) Do the same as in Exercise 17.4.5, but now assume $0 \leq N < 64$.

17.4.8.(3) Do the same as in Exercise 17.4.6, but now assume $0 \leq N < 64$.

17.4.9.(3) In mathematics it is occasionally useful to define the "integer-part-of" or "floor" function, that yields the largest integer not exceeding its argument. It is usually written with square brackets like this:

```
[X]   is the largest integer ≤ X.
```

Show that in the two's complement binary representation, the result of arithmetically right-shifting a number Z by n bit positions gives the result $[Z/(2^n)]$.

17.4.10.(3) Rewrite the code sequence of Exercise 17.3.9 assuming that the integers may be positive or negative (that is, they are stored in the arithmetic representation rather than the logical representation).

17.4.11.(2)+ Suppose there is a positive nonzero word integer stored at the word at **NUM**. Write an instruction sequence that leaves a number in GR0 that is the largest power of two less than or equal to the given number. That is, compute 2**N such that 2**N ≤ c(NUM). (For example, if c(NUM)=9, c(GR0) will be 8.)

17.4.12.(3) In Exercise 17.4.11, you wrote instructions to leave a number in GR0 that was the largest power of two less than or equal to the nonzero positive number at **NUM**. Write another instruction sequence, assuming that the number at **NUM** may be positive *or* negative. Leave a

number in GR0 that is either zero (if c(NUM) is), or is the largest power of two less than or equal to the *magnitude* of c(NUM).

17.4.13.(3) In Exercise 17.4.11, you wrote instructions to leave a number in GR0 that was the largest power of two less than or equal to the nonzero positive number in the word at **NUM**. Write another code sequence that will leave the *exponent* of that power of two in GR0. (That is, if the number left in GR0 in Exercise 17.4.11 is 2**N, c(GR0) is N.)

17.4.14.(3)+ In describing the shift instructions on page 253, it was stated that a right shift of N places was equivalent to a division by 2**N. This is sometimes true, and sometimes not true. When is it true, and when not?

17.4.15.(2) Repeat Exercise 17.3.15, assuming that the values are to be stored in the arithmetic representation.

17.4.16.(2) Write a sequence of instructions that will count the number of 1-bits in the byte at **XX** and replace the byte with its bit count.

17.4.17.(2) Suppose the initial contents of GG0 is `X'FEDCBA9876543210'` before executing *each* of these instructions:

```
(1)       SRAG  0,0,20
(2)       SLAG  0,0,28
(3)       SRA   0,18
(4)       SRLG  0,0,18
```

What result will be in GG0 after executing each instruction, and what will be the resulting CC setting?

17.4.18.(2)+ Suppose GR0 contains `X'87654321'` before executing each of these instructions. What will be in GR0 after it is executed, and what will be the CC setting?

1. `SRA  0,20`
2. `LPR  0,0`
3. `SLA  0,28`

17.4.19.(2) Suppose you want to display the individual bits in a byte at **Byte** in character form. Write a program segment that will "spread out" the bits into eight EBCDIC characters starting at **Char** so that the eight characters faithfully represent the bits in the byte.

17.4.20.(3)+ Suppose your CPU supports logical but not arithmetic shifts. Write instructions using logical shift instructions to perform the functions of SRDA, including setting the Condition Code correctly. The double-length operand to be shifted is in (GR0,GR1) and the shift amount is in GR2. Other registers may be used as needed.

17.4.21.(2)+ You can use SRA to divide a number by 2. But if the number is negative, the result isn't always what you expect. For example:

```
         L     0,=F'+5'            c(GR0) = X'00000005' = +5
         SRA   0,1                 C(GR0) = X'00000002' = +2
         L     0,=F'-5'            c(GR0) = X'FFFFFFFB' = -5
         SRA   0,1                 C(GR0) = X'FFFFFFFD' = -3
```

In both cases the result is "rounded" downward, toward −infinity. What should you do to be sure right-shifting a negative number will give the same result (except for sign) when you divide by 2 as for positive numbers?

17.4.22.(1)+ Show how you can use a shift instruction to test the sign of the contents of a general register without affecting its value.

17.4.23.(2)+ An arithmetic right shift of a binary number makes it smaller in magnitude, except for two values. What are they?

17.5. Rotating Shifts

Unlike the shift instructions we've seen, the rotating shift instructions RLL and RLLG neither lose nor introduce bits. A rotate unit shift takes the leftmost bit of the register, shifts all the other bits left one position, and inserts the previous leftmost bit at the right end of the register, as illustrated in Figure 120.

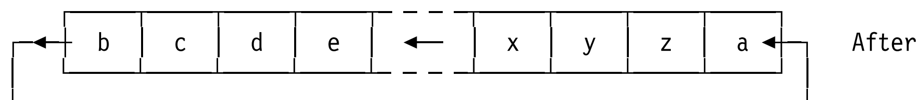


Figure 120. Logical rotate unit shift

As shown in Table 80 on page 243, the source operand in R_3 and the target operand in R_1 can be the same or different registers. If they are the same, the shift does not preserve the original operand.

The rotating shift instructions are sometimes used in data compression algorithms. In applications where speed of rotation is not important, their functions can be "emulated" using logical shifts. (See Exercises 17.5.1 and 17.5.2.)

To illustrate a rotating shift, suppose we rotate the 32-bit operand X'56789ABC' left by 10 bit positions:

```
         L     0,=A(X'56789ABC')  Load initial data into GR0
         RLL   1,0,10             Rotate 10 bits, result in GR1
```

Then c(GR1) will be X'E26AF159'. Similarly, if we rotate the 64-bit operand X'56789ABCDEF01234' left by 10 bit positions:

```
         LG    0,=AD(X'56789ABCDEF01234')   Initialize GG0
         RLLG  1,0,10             Rotate 10 bits, result in GG1
```

Then c(GG1) will be X'E26AF37BC048D159'.

Exercises

17.5.1.(2)+ Suppose your CPU has only single-length logical shift instructions (SLL, SRL). A 32-bit word at **DataWord** is to be rotated. Write an instruction sequence that simulates the RLL instruction by doing a logical rotation of N bit positions, where N is any nonnegative number stored in a halfword at **NN**. Store the result at **RotateWd**.

17.5.2.(3) Do the same as in Exercise 17.5.1, but now simulate the RLLG instruction using SLDL and SRDL to do a double-length rotating shift of N places. Assume the initial data is in a doubleword at **DWData**, and store the rotated double-length result at **RotatDWd**.

17.5.3.(1) Show how you can use a rotating shift to exchange the halves of a 64-bit general register.

17.6. Calculated Shift Amounts

As we saw in Section 17.2, the number of bit positions shifted can be specified during program execution, because the number of shifts in any shift instruction is determined from its Effective Address. For example,

```
         SLL   9,0(4)
```

will shift GR9 by an amount determined by the rightmost six bits of the contents of GR4.

Suppose GR1 contains a nonnegative integer less than 31; call it "n". Then, to leave 2^n in GR0, we could write

```
         L     0,=F'1'              Put 2**0 = 1 in GR0
         SLL   0,0(1)               Shift left 'n' places to form 2**n
```

The shift amount in GR1 could have been previously calculated or loaded into GR1 from memory.

We can use shifts to illustrate an amusing (but not recommended!) application of the USING statement. As with relocatable implied addresses, the Assembler computes displacements and assigns base registers for *absolute* implied addresses. If we write the statements below, the instructions would be assembled as indicated in the remarks fields of the last three statements.

```
         USING 6,2                  Absolute expression for base in GR2
A        EQU   10                   Symbol with absolute value
*                               Assembled instructions:
         SLL   9,12                 8990 2006 (implied address)  12 shifts
         SLL   9,12(0)              8990 000C (explicit address) 12 shifts
         SLL   9,A                  8990 2004 (implied address)  10 shifts
```

Thus we can vary the number of shifts at execution time by placing appropriate values in the “base” register, GR2. This is a *very* poor programming technique; it's far better to use an instruction like

```
         SLL   9,0(2)
```

There are very few occasions where an absolute expression is used as the first operand in a USING instruction. The need for caution is apparent when you consider what would happen to a program with the implied-address shift instructions above, and then someone changed the contents of GR2.

Exercises

17.6.1.(2)+ What will happen at *both* assembly and execution times if the following sequence of three statements appears in a program:

```
         USING *,2
A        EQU   *
         SLL   9,A
```

17.6.2.(2)+ What number of shifts is specified by

```
         SLL  9,*   ?
```

Is that number fixed within any one program?

17.6.3.(2)+ What number of shifts is specified by these instructions?

```
         SLL  9,AAA
         - - -
AAA      DC   F'12'
```

17.6.4.(2) Describe and evaluate the usefulness of each of the following methods for clearing a 32-bit general register `x` to zero: `(1) SLL  x,32 (2) L  x,=F'0' (3) LH  x,=H'0' (4) SLDL  x,32 (5) SRL  x,32 (6) SRDL  x,32 (7) SRDA  x,32 (8) SLDA  x,32.`

17.6.5.(1)+ In the mnemonics for the 32-bit (single-length) shift instructions, a consistent convention is used to indicate (1) the type, (2) the direction, and (3) the length of the shift. Make a table that displays this convention.

17.6.6.(1) Can you think of any reason to perform a logical shift of more than 31 bit positions in a single register? An arithmetic shift?

17.6.7.(2)+ We wish to generate a pair of bytes containing the EBCDIC characters corresponding to the 2 hex digits in the byte at **DATA**. That is, if c(DATA) = `X'4A'`, the generated pair of bytes will contain `X'F4C1'`. Write a code sequence that will store the two characters at `CH`

and CH+1, for any values in the byte at **DATA**. (Hint: construct a 16-byte character table, and access it with an indexed IC instruction.)

17.6.8.(2)+ Most System z instructions expect that their operands will be found in memory at addresses satisfying a specific boundary alignment. This usually means that the Effective Address of an instruction should be divisible by some number. For each of the following instructions, show the number by which the Effective Address should be divisible.

1. L
2. BC
3. LH
4. ICM
5. LR
6. SRDA
7. STM
8. STC

17.6.9.(1) How many bit positions are shifted by this instruction?

```
         SRL   7,=F'15'
```

17.7. Bit-Length Constants (*)

In Figures 117, 118, and 119 we saw examples of using shift instructions to extract and insert small binary constants in various fields within a 32-bit word. You can define constants with such lengths using *bit-length* constants.

We first encountered length modifiers for binary constants in Section 11.4 on page 140, where we defined constants like

```
         DC    FL3'8'
```

Such length modifiers determine the *byte* length of the constant.

You can also define the bit length of a constant by writing a length modifier specifying the number of bits allotted to its assembled value; follow the modifier letter L with a period and the number of bits. For example:

```
         DC    FL3'8'            can also be written
         DC    FL.24'8'
```

The same constant will be generated in both cases, aligned on the current location counter boundary (not necessarily a word boundary).

The general form of a length modifier is either

```
         LByteLength             as in L3
    or
         L(ByteLengthExpr)       as in L(2+1)
```

or

```
         L.BitLength             as in L.24
    or
         L.(BitLengthExpr)        as in L.(16+8)
```

but unfortunately you *cannot* combine the two by writing

```
         LByteLength.BitLength   as in L2.5
```

The length modifier must be either byte or bit length, not both.

For both byte- and bit-length modifiers, the length value may be written either as a positive decimal constant or as a positive absolute expression in parentheses.

A nominal value can be any length (subject to normal truncation and padding rules):

```
         DC    FL.12'2047',FL.8'64',XL.4'D'  generates X'7FF40D'
```

Incomplete bytes are padded with zero bits:

```
         DC    FL.12'2047'                   generates X'7FF0'
```

Now we can see how to generate the "packed" unsigned binary integers in Figure 115 on page 249. Suppose the four integers A, B, C, and D have values 432, 12, 5001, and 47 respectively. We can define a word containing these values as shown in Figure 121.

```
UnsdVals DC    0F,FL.9'U432',FL.4'U12',FL.13'U5001',FL.6'U47'
```

Figure 121. Packing four unsigned bit-length constants in a 32-bit word

Similarly, if the four values could be signed, with values −232, −8, −4001, and −31 respectively, we could define a word containing their values as shown in Figure 122.

```
SgndVals DC    0F,FL.9'-232',FL.4'-8',FL.13'-4001',FL.6'-31'
```

Figure 122. Packing four signed bit-length constants in a 32-bit word

Exercises

17.7.1.(1) What differences might you find for these constants?

```
A        DC    F'-97'
B        DC    FL4'-97'
C        DC    FL.32'-97'
```

17.7.2.(2)+ In Figure 121, what constant is generated? What constant would be generated if the letter "U" is omitted?

17.7.3.(2)+ In Figure 122, what constant is generated?

17.7.4.(3)+ Rewrite the constant definitions in Figures 121 and 122 to use the symbolic definitions of the four field lengths named LA, LB, LC, and LD respectively, as shown in Figure 117 on page 250.

17.7.5.(2)+ If you can't write a bit-length constant with a length modifier of the form LA.B (where A is the byte length and B is the bit length), how can you write it to achieve equivalent results?

17.8. Summary

Table 82 summarizes the shift instructions discussed in this section. As mentioned above, the notation "32+32" means that the shift is in a pair of 32-bit general registers.

Function / Operand length (bits)	32	32+32	64
Arithmetic shift	SLA SRA	SLDA SRDA	SLAG SRAG
Logical shift	SLL SRL	SLDL SRDL	SLLG SRLG
Rotating shift	RLL		RLLG

Table 82. Summary of shift instructions discussed in this section

---

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
RLL	EB1D
RLLG	EB1C
SLA	8B
SLAG	EB0B
SLDA	8F

Mnemonic	Opcode
SLDL	8D
SLL	89
SLLG	EB0D
SRA	8A
SRAG	EB0A

Mnemonic	Opcode
SRDA	8E
SRDL	8C
SRL	88
SRLG	EB0C

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
88	SRL
89	SLL
8A	SRA
8B	SLA
8C	SRDL

Opcode	Mnemonic
8D	SLDL
8E	SRDA
8F	SLDA
EB0A	SRAG
EB0B	SLAG

Opcode	Mnemonic
EB0C	SRLG
EB0D	SLLG
EB1C	RLLG
EB1D	RLL

Terms and Definitions

arithmetic shift
A movement of bits in a general register to the left or right, preserving the arithmetic sign of the operand.

logical shift
A movement of bits in a general register to the left or right, inserting zero bits into any vacated bit positions.

rotating shift
A movement of bits in a general register to the left in such a way that bits moved out of the high-order bit position are inserted into the low-order bit position. (Also called a "circulating" shift.)

Programming Problems

Problem 17.1.(2) Write a program that takes a positive word integer from the memory area named **Data** and shifts it left until its *next-to-highest-order* bit (that is, bit number 1) is nonzero. Store the result in a word area named **Norm**, and store at the halfword area **Count** the number of shifts required. Print the contents of **Data**, **Norm**, and **Count**. Run the program with several different values at **Data** such as 1, 999, 2147483647, and others.

Problem 17.2.(2) A programmer suggested using these instructions to convert the eight bits in a byte to eight EBCDIC characters representing their value.

```
         ICM   1,B'1000',DataByte Put the byte at the left end of GR1
         LH    2,=H'8'            Set the bit count to 8
Loop     SLLG  3,3,8              Make room in GG3 for the character
           SR    0,0              Clear GR0
           SLDL  0,1              Shift a low-order bit into GR0
           A     0,=A(X'F0')      Add X'F0' to make a character
           ALR   3,0              Insert the character into GG3
         SH    2,=H'1'            Count down by 1
         BP    Loop               Repeat for all 8 bits
         STG   3,BitChars         Store the 8 characters
         - - -
BitChars DS    D                  8 EBCDIC 0 and 1 characters
```

Write a program with several data values to test her assertion.

Problem 17.3.(1) Using the instructions in Figure 119 on page 254, write a program to unpack the four signed integers of Figure 122 on page 260 at the word named **SgndVals** and display the unpacked values at **First**, **Second**, **Third**, and **Fourth** as fullword integers.

18. Binary Multiplication and Division

```
    11      8888888888
   111     888888888888
  1111     88        88
    11     88        88
    11       88888888
    11       88888888
    11     88        88
    11     88        88
    11     88        88
1111111111 888888888888
1111111111  8888888888
```

When we multiply two numbers, the product can be as long as the sum of their lengths. For example, multiplying the three-digit decimal number 999 by itself, 999×999 gives 998001: six digits long. Thus, we will need double-length registers if our products of single-length numbers can be longer than a single register.

The terminology used for the operands is from mathematics:

```
  multiplicand  (first operand)
×   multiplier  (second operand)
       product
```

18.1. Overview of Multiplication Instructions

The instructions we'll examine are summarized in Table 83. The notation "32×32" means the product of two 32-bit integers, and similarly for "32×16", "64×64", and "64×32".

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5C	M	RX	Multiply (32+32←32×32)	1C	MR	RR	Multiply Register (32+32←32×32)
4C	MH	RX	Multiply Halfword (32←32×16)				
71	MS	RX	Multiply Single (32←32×32)	B252	MSR	RRE	Multiply Single Register (32←32×32)
E351	MSY	RXY	Multiply Single (32←32×32)				
E30C	MSG	RXY	Multiply Single (64←64×64)	B90C	MSGR	RRE	Multiply Single Register (64←64×64)
E31C	MSGF	RXY	Multiply Single (64←64×32)	B91C	MSGFR	RRE	Multiply Single Register (64←64×32)
E396	ML	RXY	Multiply Logical (32+32←32×32)	B996	MLR	RRE	Multiply Logical Register (32+32←32×32)
E386	MLG	RXY	Multiply Logical (64+64←64×64)	B986	MLGR	RRE	Multiply Logical Register (64+64←64×64)

Table 83. Binary integer multiply instructions

The result of each multiply instruction is a 32-bit, 64-bit, or 128-bit product, as indicated by "32←..." (for a single 32-bit register), "32+32←..." (for a 64-bit product in a pair of 32-bit registers), "64←..." (for a single 64-bit register), and "64+64←..." (for a 128-bit product in a pair of 64-bit registers). As we saw for signed and logical addition and subtraction, signed multiplications sign-extend short operands, and logical multiplications zero-extend short operands.

As Table 83 on page 264 indicates, there are no instructions giving a 128-bit *arithmetic* product of two signed 64-bit operands.[112]

None of these instructions change the CC setting.

Condition Code

Binary multiplication and division do not change the CC setting.

Signed multiply instructions are the most frequently used, so we'll discuss them first.

18.2. Arithmetic (Signed) Multiplication Instructions

The two types of arithmetic multiplication instructions give either single-length or double-length products. Because double-length products are more often used, we'll start with those.

18.2.1. Double-Length Arithmetic Products

The instructions yielding arithmetic 64-bit double-length products are:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5C	M	RX	Multiply (32+32←32×32)	1C	MR	RR	Multiply Register (32+32←32×32)

Table 84. Double-length arithmetic multiply instructions

M and MR form the 64-bit product of two 32-bit operands. The first operand, the multiplicand, is in the odd-numbered register of an even-odd register pair. The second operand, the multiplier, is either in a register or a word in memory, as illustrated in Figure 123. Note that the initial contents of the even-numbered register, GR R_1, are ignored (unless GR R_1 contains the second operand).

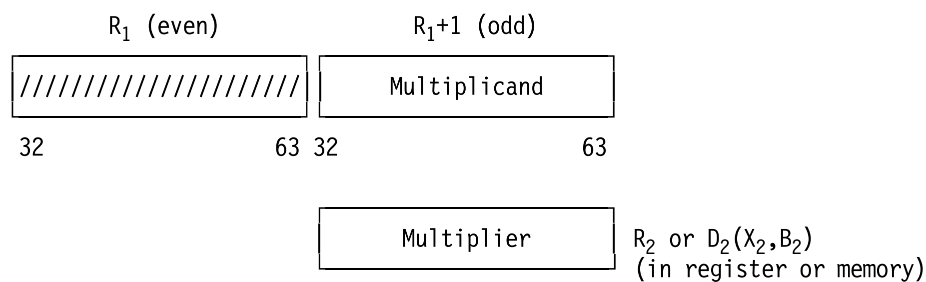


Figure 123. General layout of multiplication operands

After the operation completes, the 64-bit product is in the register pair, as shown in Figure 124 on page 266.

[112] At the time of this writing. But new instructions are added regularly to the System z architecture, so check the *Principles of Operation*. However, you can generate signed products using the unsigned multiply instructions; see Exercises 18.3.2 and 18.3.3.

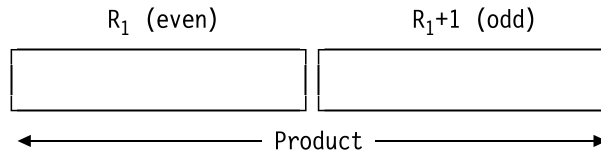


Figure 124. Double-length product of multiply operations

For M and MR, no fixed-point overflow is possible. As with the double-length shift instructions, the even-numbered register is the high-order half of an even-odd register pair, and the next higher odd-numbered register is the low-order half. The CPU takes the multiplicand from the odd-numbered register and the multiplier from the address or register specified by the second operand. The product replaces the original contents of the pair of registers, and the high-order bit of the odd-numbered register is a part of the product, not necessarily a sign bit. The following instructions produce the indicated results.

```
         MR   2,7               c(GR2,GR3) = c(GR3) * c(GR7)
*   Square the number in GR1
         MR   0,1               c(GR0,GR1) = c(GR1) * c(GR1)
         MR   8,8               c(GR8,GR9) = c(GR9) * c(GR8)
         M    4,XX              c(GR4,GR5) = c(GR5) * c(XX)
         M    12,=F'932'        c(GR12,GR13) = c(GR13) * 932
*   Square the number in GR4
         LR   5,4               Move multiplicand to GR5
         MR   4,4               c(GR4,GR5) = c(GR5) * c(GR4)
```

The last two instructions show how to square the integer in GR4: the LR instruction copies the multiplier to the odd-numbered register. The presence of the multiplier in the even-numbered register does not cause it to be lost when that register is cleared at the beginning of the multiply sequence; the multiplication takes place *after* the CPU has saved a copy of the multiplier. After the LR we could also have used “MR 4,5”, giving c(GR5)×c(GR5).

The product generated by the M and MR instructions is 64 bits long. If we perform these instructions:

```
         L     1,=A(X'10000')   c(GR1) = 65536 = 2**16
         MR    0,1              Square it to get 2**32
         ST    1,Product        Store low-order half
         - - -
Product  DS    F
```

we would find that the word stored at **Product** was zero, and that c(GR0) = 1. Similarly, if we execute these instructions (where 32768 = 2^{15}):

```
         L     1,=A(X'10000')   c(GR1) = 65536
         M     0,=A(X'8000')    Multiply by 32768; result = +2**31
         ST    1,Product        Store +2**31 (??)
```

we would find that c(GR0)=0, and c(Product) = -2^{31}!

There are two situations needing caution. First, the product may be so long that significant bits occupy more than the low-order register. Second, whether or not the high-order register contains significant bits, the leftmost bit of the low-order register can be interpreted as a sign bit *only* if the product lies in the range

$$-2^{31} \le \text{product} < +2^{31}$$

Otherwise, the low-order sign bit contains an arithmetically significant digit with positive weight.

As an example using a multiply instruction, suppose we want to evaluate **A = B + G * D**, a typical expression in a high-level language. All quantities are word integers, and we assume all results are small enough so that no overflows occur.

```
        L     7,G                 c(GR7) = c(G)
        M     6,D                 c(GR6,GR7) = G * D
        A     7,B                 c(GR7) = B + (G*D)
        ST    7,A                 Store result at A
```

We have used the symbols **A**, **B**, **G**, and **D** to denote both the names of word areas of memory and the values of the contents of those areas (that is, as “variables”). This usage is typical of high-level languages, where little distinction is made among the name associated with an area of memory, the contents of that area, the value associated with the contents, and the name of the value.[113]

Suppose we wish to compute the sum of the cubes of the first N integers, where N is stored in the word at **NBR**. We assume that N is a small enough positive integer that the sum of the cubes is representable in a single word. The quantity called “K” is a counter that runs from 1 to N in steps of 1.

```
         SR    5,5                Sum carried in GR5
         L     4,=F'1'            Initialize K in GR4
Repeat   LR    1,4                c(GR1) = K
           MR    0,1              c(GR0,GR1) = K * K
           MR    0,4              c(GR0,GR1) = K cubed
           AR    5,1              Accumulate sum
           A     4,=F'1'          Increment K
           C     4,NBR            Compare to upper limit at NBR
         BNH   Repeat             Repeat if K is not bigger
         ST    5,Sum              Store sum of first N cubes
         - - -
NBR      DC    F'10'              N
```

Figure 125 shows a slightly different version of this example; it counts from N down to 1:

```
         SR    5,5                initialize sum to zero
         L     4,NBR              Initialize K from c(NBR) = N
Repeat   LR    1,4                c(GR1) = K
           MR    0,4              c(GR0,GR1) = K * K
           MR    0,4              c(GR0,GR1) = K cubed
           AR    5,1              Add to sum
           S     4,=F'1'          Decrement K by 1
         BP    Repeat             Repeat if K is still positive
         ST    5,SUM              Store sum of first N cubes
```

Figure 125. Calculate the sum of the first 10 cubed integers

18.2.2. Single-Length Arithmetic Products

The instructions generating single-length arithmetic products are shown in Table 85 on page 268.

When you know a product will be small enough to fit correctly in a single-length register, or if you don't care that some high-order bits may be lost, these instructions avoid needing an even-odd register pair, and may also execute faster than the instructions generating double-length products.

[113] These distinctions are *very* important in Assembler Language, and can be very confusing to people whose first programming experiences were with high-level languages.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
4C	MH	RS	Multiply Halfword (32←32×16)				
71	MS	RX	Multiply Single (32←32×32)	B252	MSR	RRE	Multiply Single Register (32←32×32)
E351	MSY	RXY	Multiply Single (32←32×32)				
E30C	MSG	RXY	Multiply Single (64←64×64)	B90C	MSGR	RRE	Multiply Single Register (64←64×64)
E31C	MSGF	RXY	Multiply Single (64←64×32)	B91C	MSGFR	RRE	Multiply Single Register (64←64×32)

Table 85. Single-length arithmetic multiply instructions

The MH instruction produces a *single-length* (word) result, the low-order 32 bits of the product of c(GR R_1) and the halfword second operand. Because only a word result is retained, R_1 need not be even. For example,

```
         MH    5,=H'100'          Multiply c(GR5) by 100
```

is a simple way to multiply the contents of GR5 by 100 without affecting the contents of the lower even-numbered register, GR4. If **X** and **Y** are both halfword operands, their product may be found by writing

```
         LH    8,X                Multiplicand in GR8 (even register!)
         MH    8,Y                Multiply by c(Y), product in GR8
```

and GR9 remains undisturbed. To square the halfword integer at **N**, we could write

```
         LH    1,N               c(N) in GR1
         MH    1,N               N squared in GR1
```

Because both operands are halfwords with at most 15 significant bits, the product will always fit in a single register. The only halfword whose magnitude requires 16 bits (-2^{15}) when squared yields 2^{30}, requiring only 31 bits.

As we've seen for MH, all the "Multiply Single" instructions place the product in a single-length register. The register may be either even- or odd-numbered; the other register of the pair is not changed. Other instructions generating a product in a 32-bit register are MS and MSR. For example:

```
         L     1,=F'12345'        c(GR1) = 12345
         MS    1,=F'12347'        c(GR1) = 152423715

         L     1,=F'12345'        c(GR1) = 12345
         L     7,=F'12347'        c(GR7) = 12347
         MSR   1,7                c(GR1) = 152423715
```

and the product is small enough to be held correctly in GR1.

MSG and MSGR, and MSGF and MSGFR produce a 64-bit product in a single 64-bit register. MSG and MSGR are exact analogs of MS and MSR:

```
         LG    1,=FD'12345678'    c(GG1) = 12345678
         MSG   1,=FD'23456789'    c(GG1) = 289589963907942

         LG    1,=FD'12345678'    c(GG1) = 12345678
         LG    7,=FD'23456789'    c(GG7) = 23456789
         MSGR  1,7                c(GG1) = 289589963907942
```

MSGF and MSGFR generate a 64-bit product of a 64-bit first operand and a 32-bit second operand by first internally sign-extending the 32-bit second operand to 64 bits:

```
LG    1,=FD'12345678'    c(GG1) = 12345678
MSGF  1,=F'23456789'     c(GG1) = 289589963907942

LG    1,=FD'12345678'    c(GG1) = 12345678
L     5,=F'23456789'     c(GR5) = 23456789 (32 bits!)
MSGFR 1,5                c(GG1) = 289589963907942
```

Exercises

18.2.1.(1)+ What is the value of the largest 64-bit product that can be generated by signed multiplication of 32-bit operands?

18.2.2.(4) Given two unsigned 32-bit integers stored in the words at **X** and **Y**, show first how you can generate their *unsigned* 64-bit product using the arithmetic multiplication instructions M and MR. Then, write a sequence of instructions that will store the product in the doubleword at **LogProd**.

Let X be the logical (unsigned) representation corresponding to the arithmetic representation x of some integer, and similarly for Y and y. To form the *logical* product of the operands X and Y, we must modify the product xy given by the processor operation of multiplication, which assumes that the operands are in the arithmetic representation. It will help to remember (from Section 2.7) that

$$XY = (2^{32}+x)(2^{32}+y) = 2^{64} + 2^{32}(x+y) + xy \text{ (modulo } 2^{64})$$

18.2.3.(2)+ What is the value of the largest 48-bit product that can be generated by signed multiplication of 32-bit and 16-bit operands? The largest 96-bit value generated by signed multiplication of 32-bit and 64-bit operands?

18.2.4.(3)+ Write a sequence of instructions that forms the product of the positive word integers at **A** and **B**, leaves the result in (GR0,GR1), and transfers to **Overflow** if the result is too large to be represented in a word.

18.2.5.(4)+ Do the same as in Exercise 18.2.4, but make no restrictions on the signs of the operands.

18.2.6.(4)+ Rewrite your solution to Exercise 18.2.5 to branch to **OverPos** if the result is too large and positive, and to **OverNeg** if the result is too large and negative.

18.2.7.(3) Suppose GR11 and GR12 contain the addresses of the first items in two tables of ten consecutive halfword integers each. Write a code sequence that computes the "inner product" of the two tables; that is, compute the product of the first elements from each table, add to it the product of the second items, etc. Store the final sum as a double-length integer beginning at the word named **DwSum**. The addresses in R11 and R12 may be modified. Since there are ten products, the accumulated sum could overflow the capacity of a single register. Be sure to handle negative products correctly.

18.2.8.(3) Simplify the coding of Exercise 18.2.2 assuming that the arithmetic representation corresponding to x is known to be positive at all times.

18.2.9.(2) When we use the M and MR instructions, the first operand specifies an even-numbered register. However, the multiplicand is actually in the next higher odd-numbered register. Can you think of any reasons why the designers of System z did not require that the actual (odd) multiplicand register be specified?

18.2.10.(2)+ Write a simple sequence of instructions that will determine whether the 64-bit product in (GR0,GR1) is too large to be carried in a single register.

18.2.11.(2)+ If all values are positive, what is the value of the largest 48-bit product that can be generated by multiplication of 32-bit and 16-bit operands? The largest 96-bit value generated by multiplication of 32-bit and 64-bit operands?

18.2.12.(3) Given a signed 32-bit operand A and an unsigned 32-bit operand B, write instructions that will generate their signed 64-bit product.

18.2.13.(2)+ A programmer wanted to test whether the product of two positive 32-bit binary integers was too large to fit in a 32-bit register. Will these instructions do what he wants?

```
         L     1,X                Load first operand
         M     0,Y                Multiply by second operand
         LTR   0,0                Check high-order 32 bits
         BZ    ProdOK             If they're zero, product fits
         - - -                    Not OK
X        DC    F'...'
Y        DC    F'...'
```

18.2.14.(3)+ You have created a signed binary product in (GR0,GR1) using instructions like

```
         L     1,X
         M     0,Y
```

and you want to determine whether its value can be stored correctly in the 32-bit field **Prod32** or (to be stored correctly) must be stored in the 64-bit field **Prod64**. Write instructions to make that determination and store the result.

18.2.15.(2) What would be stored at Z by these instructions?

```
         L     3,X
         M     2,Y
         SR    2,3
         ST    2,Z
         - - -
X        DC    F'9'
Y        DC    F'-7'
Z        DS    F
```

18.2.16.(1)+ Suppose these two instructions are executed:

```
         LH    1,N                Get halfword from N
         MSR   1,1                Square it
```

Show the value that will be in GR1 if the number at **N** is (1) X'8000' and (2) X'FFFF'.

18.3. Logical (Unsigned) Multiplication Instructions

Table 86 lists the logical multiplication instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E396	ML	RXY	Multiply Logical (32+32←32×32)	B996	MLR	RRE	Multiply Logical Register (32+32←32×32)
E386	MLG	RXY	Multiply Logical (64+64←64×64)	B986	MLGR	RRE	Multiply Logical Register (64+64←64×64)

Table 86. Logical multiply instructions

Logical multiply instructions are similar to arithmetic multiply instructions, except that the operands and results are unsigned. All four instructions generate a double-length product in an even-odd register pair. Logical multiplication is frequently used when high- or multiple-precision calculations are required[114]. Although you can use arithmetic multiplication instructions to generate logical products, and logical multiplication instructions to generate arithmetic products, extra

[114] Some encryption and decryption algorithms use multiple-precision arithmetic extensively.

instructions and time are needed.[115] It's simplest to use whichever instruction is best suited to the type of operand.

For example, suppose you multiply the maximum negative 32-bit number by itself using arithmetic and logical multiply instructions:

```
*       Arithmetic multiplication
        L     1,=X'80000000'     c(GR1) = -2147483648
        MR    0,1                c(GR0,GR1) = X'40000000 00000000'

*       Logical multiplication
        L     1,=X'80000000'     c(GR1) = +2147483648
        MLR   0,1                c(GR0,GR1) = X'40000000 00000000'
```

The result is the same in both cases. Arithmetically, the maximum negative number has value -2^{31}, and the same bit pattern as an unsigned number has value $+2^{31}$. Thus, the product in both cases is $+2^{62}$. Now, let's try squaring a different operand, -1:

```
*  Arithmetic multiplication: (-1)*(-1) = +1
        L     1,=F'-1'           c(GR1) = X'FFFFFFFF'
        MR    0,1                c(GR0,GR1) = X'00000000 00000001'

*  Logical multiplication: (2**32-1)*(2**32-1) = 18446744065119617025
        L     1,=F'-1'           c(GR1) = X'FFFFFFFF'
        MLR   0,1                c(GR0,GR1) = X'FFFFFFFE 00000001'
```

These results are *very* different! The bit pattern `X'FFFFFFFF'` represents –1 arithmetically, but $2^{32}-1$ logically.

The MLG and MLGR instructions generate 128-bit products in an even-odd pair of 64-bit registers:

```
        LG    1,=FD'74296604373' c(GG1) = 74296604373
        MLG   0,=FD'9876543210'  c(GG0,GG1) = 733793623446209457330

        LG    1,=FD'74296604373' c(GG1) = 74296604373
        LG    3,=FD'9876543210'  c(GG3) = 9876543210
        MLGR  0,3                c(GG0,GG1) = 733793623446209457330
```

These instructions can generate very large products!

Exercises

18.3.1.(1) What is the value of the largest 64-bit product that can be generated by logical multiplication of 32-bit operands?

18.3.2.(3) Given two signed 32-bit integers stored in the words at **P** and **Q**, show first how you can generate their *signed* 64-bit product using the logical multiplication instructions ML and MLR. Then, write a sequence of instructions that will store the product in the doubleword at **ArProd**.

18.3.3.(4) Do the same as in Exercise 18.3.2, but this time form the 128-bit signed product of two 64-bit signed operands at **DP** and **DQ** using the logical multiplication instructions MLG and MLGR. Store the result in the pair of doublewords at **ArProd2**.

18.3.4.(4) As in Exercise 16.6.3 on page 229, form the product of the two 256-bit integers at **A256** and **B256** to form a 512-bit product stored at **Prod256**.

[115] Try Exercises 18.2.2 and 18.3.2.

18.4. How Multiplication Is Done (*)

To illustrate the method used in multiplication, we'll first use an example in decimal arithmetic. Suppose we have a “processor” with registers that hold 3-digit decimal numbers that we assume are positive, and we multiply 213 and 126. Since we are multiplying two 3-digit numbers, the product can be 6 digits long. Thus, we assume there is a double-length 6-digit register whose right and left halves hold a 3-digit number.

When working with pencil and paper, we form the product of the multiplier and each of the multiplicand digits in succession, and generate a series of partial products that must be properly aligned and then added:

```
Multiplicand      213
Multiplier      × 126
partial          1278
products         426
                213
Product         26838
```

We'll now see how this manual process can be broken down into steps that are more like the method used in a computer.

1. We place the multiplicand in the *right* half of the double-length register, and clear the left half to zero.

```
Initial register contents         000 213
```

2. By examining the rightmost digit of the multiplicand we know how many times to add the multiplier to the *left* half of the double-length register. As an aid in counting how many times to add the multiplier, we decrement the rightmost multiplicand digit by 1 for each addition. When the rightmost digit has been counted down to zero, the partial product of that digit and the multiplier has been added to the accumulating result.

```
Initial register contents         000 213
Add multiplier to upper end      +126
  that's 1 time                   126 212, count down at right
Add multiplier                   +126
  that's 2 times                  252 211, count down at right
Add multiplier                   +126
  that's 3 times                  378 210, count down at right
```

3. The entire double-length register is shifted right one digit position, at which time the (now) zero digit at the right-hand end is lost, and a zero digit is inserted in the vacated position at the left.

```
Shift right one place             037 821
Add multiplier                   +126
  that's 1 time                   163 820, count down at right
```

4. After the second shift, the final multiplicand digit is 2:

```
Shift right one place             016 382
Add multiplier                   +126
  that's 1 time                   142 381, count down at right
Add multiplier                   +126
  that's 2 times                  268 380, count down at right
Shift right one place             026 838
```

This process of adding the multiplier and counting down on the multiplicand digit continues until the proper partial product has been added to the accumulated result. This process is repeated for as many steps as there are multiplicand digits. When completed, the product is in the double-length register, and all multiplicand digits have been shifted off the right-hand end.

The main points are:

- the multiplicand is initially placed in the right half of the double-length register;

- the left half is initially cleared to zero (after saving the multiplier if it was in the left half);
- the multiplier is added to the left end a number of times determined by the multiplicand digit at the far right; and
- the least significant digit of the result is at the right-hand end of the *double-length* register, because the number of right shifts was the same as the number of positions in a single-length register.

When used for multiplying *binary* numbers, the above scheme is very easy to implement, because testing the rightmost bit determines whether or not the multiplier is to be added, and no counting is required. Suppose we have 5-digit binary numbers and registers, and wish to multiply B'00110' (=6) by B'01001' (=9) to obtain a 10-bit product in a double-length register. The sequence of steps in Figure 126 shows how this is done.

```
Initialize                   00000 01001  Multiplicand, in right half of
                                            double-length register

                             00110        Multiplier, in separate register

Step 1: Rightmost bit = 1,
        Add multiplier       00110 01001
        Shift right 1 place  00011 00100  (The 1-bit is lost)
Step 2: Rightmost bit = 0,
        Shift right 1 place  00001 10010
Step 3: Rightmost bit = 0,
        Shift right 1 place  00000 11001
Step 4: Rightmost bit = 1,
        Add multiplier       00110 11001
        Shift right 1 place  00011 01100  (The 1-bit is lost)
Step 5: rightmost bit = 0,
        Shift right 1 place  00001 10110  Final product (=54)
```

Figure 126. Illustration of binary multiplication

It is important to observe that the product really is a double-length number, and not just two single-length numbers joined end to end. If we consider the contents of the left and right halves of the double-length register as ordinary single-length two's complement operands, we might believe the result in the right, or low-order half, was negative! Since a product of two positive numbers must be positive, a double-length register means that no special significance can be attached to the sign bit of the low-order half of the result, unless we know in advance that the product is correctly representable in a single register.[116] The leftmost bit of the right-hand register is therefore *not* a sign bit; it has positive weight in the double-length result, and the product's sign bit is the leftmost bit of the high-order register.

Modern processors gain speed by considering not just the rightmost bit of the multiplicand, but groups of two, three, or even four bits. In cases where the arithmetic can be considered to be base 4, 8, or 16, the “proper multiple” is not found by counting down by ones on the multiplicand bits, but by having internal shifting or table look-up circuits generate the proper factor of the multiplier in many fewer steps. This increases the speed of multiplication, since a separate addition is not required for each 1 bit in the multiplicand.

[116] Because many multiplications involve small numbers not needing a double-length product, the various “Multiply Single” instructions were created. They can be faster than instructions generating double-length products.

18.5. Division Instructions

As with multiplication, the terminology is taken from mathematics:

```
          Quotient
Divisor ) Dividend
           - - - -
         Remainder
```

While multiplying two n-digit numbers usually gives a 2n-digit product, dividing a 2n-digit dividend by an n-digit divisor does not necessarily produce an n-digit quotient. If for example we use 3-digit decimal numbers, 999×999=998001; but 998001÷100 gives quotient 9980 and remainder 1, with a 4-digit quotient.

If the divisor is zero, or if the quotient is too large to fit in a single-length register, a *Fixed-Point Divide interruption* will occur, with Interruption Code 9. This condition cannot be suppressed (as can Fixed-Point Overflow). It is important to be careful when preparing for division!

The divide instructions we'll consider are shown in Table 87.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5D	D	RX	Divide (32,32←32+32÷32)	1D	DR	RR	Divide Register (32,32←32+32÷32)
E30D	DSG	RXY	Divide Single (64,64←64÷64)	B90D	DSGR	RRE	Divide Single Register (64,64←64÷64)
E31D	DSGF	RXY	Divide Single (64,64←64÷32)	B91D	DSGFR	RRE	Divide Single Register (64,64←64÷32)
E397	DL	RXY	Divide Logical (32,32←32+32÷32)	B997	DLR	RRE	Divide Logical Register (32,32←32+32÷32)
E387	DLG	RXY	Divide Logical (64,64←64+64÷64)	B987	DLGR	RRE	Divide Logical Register (64,64←64+64÷64)

Table 87. Binary divide instructions

The notation describing the operands and results of these instructions shows the general register results to the left of the "←" character, and the dividend and divisor to the right. For example, for the D instruction, (32,32←32+32÷32) means that the quotient and remainder 32,32 are both 32-bit words; the dividend 32+32 is a pair of 32-bit registers, and the divisor is a 32-bit integer. Similarly, for DSGF, (64,64←64÷32) means that the quotient, remainder, and dividend are 64-bit integers, and the divisor is a 32-bit sign-extended integer.

As Table 87 indicates, there are no instructions (like DG, DGR) for dividing 128-bit arithmetic operands by a signed 64-bit divisor of the form (64,64←128÷64), nor instructions (like DS, DSR) for dividing 32-bit signed operands by 32-bit divisors.[117]

When any division instruction completes without interruption, the quotient is found in the odd-numbered register of the pair, and the remainder in the even-numbered register, as illustrated in Figure 127.

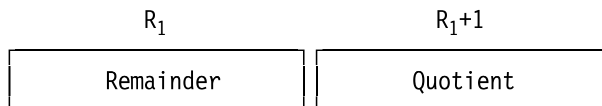


Figure 127. General result of divide operation

[117] At the time of this writing. But new instructions are added regularly to the System z architecture, so check the *Principles of Operation*.

None of the divide instructions changes the CC setting, and an even-odd register pair is always required, even for the "Divide Single" instructions.

Register Pairs for Division

All System z binary integer divide instructions require an even-odd register pair.

Exercises

18.5.1.(2)+ If you divide an ND-digit dividend (numerator) by a DD-digit divisor (denominator), what are the minimum and maximum numbers of digits QD in the quotient and RD in the remainder? Assume a valid division, and that zero is a valid result.

18.6. Arithmetic (Signed) Division Instructions

Table 88 summarizes the arithmetic division instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
5D	D	RX	Divide (32,32←32+32÷32)	1D	DR	RR	Divide Register (32,32←32+32÷32)
E30D	DSG	RXY	Divide Single (64,64←64÷64)	B90D	DSGR	RRE	Divide Single Register (64,64←64÷64)
E31D	DSGF	RXY	Divide Single (64,64←64÷32)	B91D	DSGFR	RRE	Divide Single Register (64,64←64÷32)

Table 88. Arithmetic divide instructions

The Divide Single instructions (DSG, DSGR, DSGF, and DSGFR) have only a single-length dividend; we'll examine them shortly.

18.6.1. Double-Length Division

The most commonly used divide instructions are D and DR. The 64-bit double-length dividend (the first operand) is placed in an even-odd pair of 32-bit registers, and the second operand (the divisor) is in another register or a word in memory. This is illustrated in Figure 128.

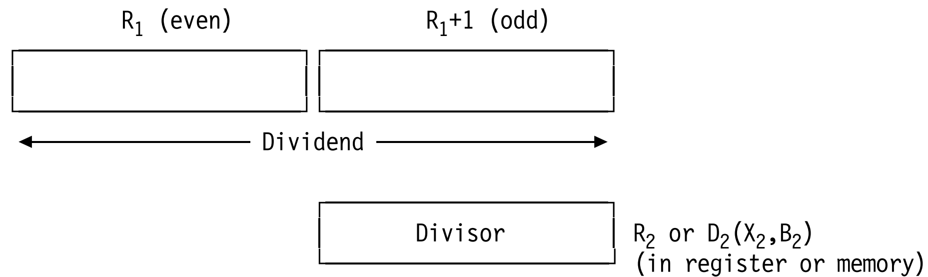


Figure 128. Operands of double-length division

This type of division uses a double-length dividend and a single-length divisor, yielding single-length quotient and remainder. The sign of the quotient is determined from the usual rules of algebra; the sign of the remainder is the same as the sign of the original dividend, except that a zero quotient or remainder always has a zero sign bit.

As with the double-length multiply instructions, the R_1 digit is always even, and specifies the register pair containing the double-length dividend. The quotient replaces the low-order half of the dividend in the odd-numbered register, and the remainder replaces the high-order part of the dividend in the even-numbered register. If a valid quotient cannot be computed, a Fixed-Point Divide interruption occurs. (An improper division is shown in Figure 133 on page 277.)

To illustrate, we divide the double-length number in (GR8,GR9) by the number in GR13.

```
         DR   8,13                Divide c(GR8,GR9) by c(GR13)
```

To divide the same number by 10 we could write

```
         D    8,=F'10'            Divide c(GR8,GR9) by 10
```

The most common use of division occurs when dividing a 32-bit word operand by another. For double-length dividends that must be 64 bits long, you can't just load the dividend operand into an odd-numbered register and immediately divide, because the even-numbered register is treated by the CPU as containing the most significant bits of the dividend. We must first *extend* the sign bit of the single-length dividend to form its correct double-length representation.

There are two ways to do this:

1. Multiply the 32-bit dividend (in the odd-numbered register) by 1:

```
         L     7,NN                Load 32-bit dividend in GR7
         M     6,=F'1'             Times 1 gives 64-bit signed dividend
```

   While easy to understand, this method may be slower than the next.

2. The most common method is to load the 32-bit dividend into the even-numbered register, and then use an SRDA instruction:

```
         L     6,NN                c(GR6) = c(NN)
         SRDA  6,32                c(GR6,GR7) = 64-bit signed dividend
```

Suppose we want to divide the positive or negative word integer at **G** by three, and store the quotient at **G_Over_3**.

```
         L    8,G                 Put numerator into even register
         SRDA 8,32                Sign-extend to double length
         D    8,=F'3'             Divide by three
         ST   9,G_Over_3          Store quotient
```

Figure 129. Example of division by 3

Suppose we want to compute the product of the integers in the words named **A** and **B** and force the result to the next larger multiple of 29 if it is not already an exact multiple. (We assume that the product is small enough that a fixed-point divide interruption will not occur when dividing by 29, and that the final result fits in a single word.)

```
         L    3,A                 c(GR3) = c(A)
         M    2,B                 c(GR2,GR3) = c(A) * c(B)
         D    2,=F'29'            Quotient in GR3
         LTR  2,2                 Test remainder in GR2
         BZ   Mult                Branch if c(GR2) is zero
         A    3,=F'1'             increase quotient by 1
Mult     M    2,=F'29'            Form correct multiple of 29
         ST   3,Result            Store proper result
```

This example assumes the final product is correctly represented in the 32 bits of GR3.

Here are two examples of division with rounding.

1. Suppose we want to divide the positive integer at **NN** by 10, and store the *rounded* quotient at **QQ**. This means that if the remainder is 5 or larger, the quotient must be increased by 1.

```
         L    7,NN                Low-order part of positive dividend in GR7
         SR   6,6                 Set high-order part to zero
         D    6,=F'10'            Divide by 10
         C    6,=F'5'             Compare remainder to 5
         BL   NoRound             Branch if smaller than 5
         A    7,=F'1'             Otherwise round up
NoRound  ST   7,QQ                Store rounded result
```

Figure 130. Example of rounded integer division

2. Now, suppose the integer at **NN** can be either positive *or* negative. The above instruction sequence will not work, for two reasons. First, the initial value of the dividend would not have a correctly extended sign bit for negative arguments (because we used SR to set the high-order register to zero). Second, because the sign of the remainder is always the same as the sign of the original dividend, if c(NN) is negative the compare instruction will always cause the following branch instruction to transfer control to **NoRound**, independent of the magnitude of the remainder.

   Here's an example of rounding the quotient of a signed dividend:

```
        L     1,=F'1'           Set up rounding increment
        L     6,NN              c(GR6) = c(NN)
        SRDA  6,32              c(GR6,GR7) = 64-bit signed dividend
        BNM   Divide            Jump if nonnegative dividend
        LCR   1,1               Otherwise set roundoff to -1
Divide  D     6,=F'10'          Divide by 10
        LPR   6,6               Take magnitude of remainder
        C     6,=F'5'           Compare to 5
        BL    NoRound           Branch if smaller than 5
        AR    7,1               Add correctly-signed roundoff
NoRound ST    7,QQ              Store rounded quotient
```

Figure 131. Example of rounded integer division with signed dividend

See Exercise 18.6.13 for a more general technique for calculating a rounded quotient.

A simple check can be made to ensure that a fixed-point divide interruption does not occur: if the inequality

$$|\text{dividend}| < |\text{divisor}| * 2^{31}$$

Figure 132. Ensuring a valid arithmetic division

is satisfied, then the quotient will be computed correctly. If an equality occurs in comparing these two quantities, we must also check for the possibility that the quotient might be exactly equal to -2^{31}.

To illustrate this relationship, suppose we want to divide the double-length dividend

X'0000000100000000' = 2^{32}

by two. Comparing dividend and divisor, the dividend might appear to be small enough to produce a valid quotient:

X'0000000100000000' = 2^{32} (dividend)
X'00000002'   = 2   (divisor; high-order part of dividend is smaller?)

The divisor 2 multiplied by 2^{31} is actually *equal* to the dividend, so that the inequality in Figure 132 is not satisfied. Since both dividend and divisor are positive, the quotient must also be positive; but the quotient is actually X'80000000', which is not representable as a positive number for signed division.

Thus, a fixed-point divide interruption can be thought of as indicating a "quotient overflow". To show how this might occur in a program, consider the segment below.

```
        L     1,=A(X'40000')    c(GR1) = 2**18
        MR    0,1               Square it, to generate 2**36
        D     0,=F'10'          Try to divide by 10
```

Figure 133. Causing a fixed-point divide interruption

Because 2^{36} is not less than 10×2^{31}, a fixed-point divide interruption will occur.

18.6.2. Single-Length Division

The arithmetic division instructions using a single-length dividend in a 64-bit register are DSG, DSGR, DSGF, and DSGFR. Even though the dividend occupies a single 64-bit register (unlike double-length dividends that require a register pair), a single-length dividend is *always* placed in the odd-numbered register. (It's easiest to think of it as being extended internally to double length before division begins.)

Even though the dividend is in the odd-numbered register, the instruction must specify the even-numbered register as the R_1 operand. This is illustrated in Figure 134 on page 278.

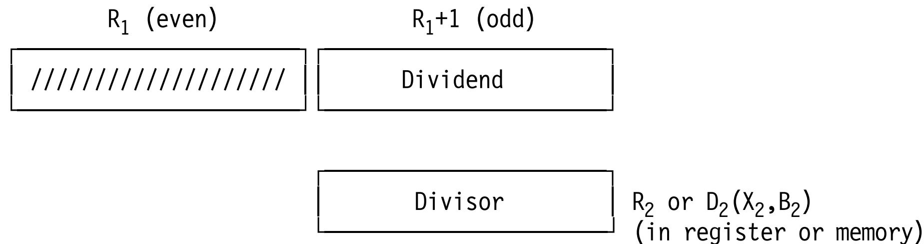


Figure 134. Operands of single-length division before division

After division, the results appear as in Figure 135.

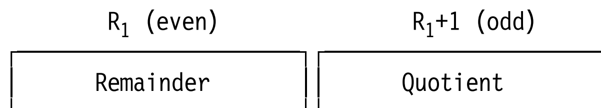


Figure 135. Operands of single-length division after division

For example, suppose you want to divide 12345678901 by 777:

```
         LG    5,=FD'12345678901' c(GG1) = 12345678901
         DSG   4,=FD'777'         Divide by 777 (64-bit divisor)
*  c(GG4) = 493 (remainder), c(GG5) = 15888904 (quotient)

         LG    5,=FD'12345678901' c(GG1) = 12345678901
         LG    9,=FD'777'         c(GG9) = 777 (64 bits)
         DSGR  4,9                Divide by 777
```

The same divisions using DSGF and DSGFR with 32-bit divisors are very similar:

```
         LG    5,=FD'12345678901' c(GG1) = 12345678901
         DSGF  4,=F'777'          Divide by 777 (32-bit divisor)

         LG    5,=FD'12345678901' c(GG1) = 12345678901
         L     9,=F'777'          c(GR9) = 777 (32 bits)
         DSGFR 4,9                Divide by 777
```

and the 32-bit second operands are internally sign-extended to 64 bits.

Note that for single-length division, there is no need to initialize the even-numbered register R_1.

Exercises

18.6.1.(2) In the inequality in Figure 132 on page 277 that assures that a division will be correct, explain the factor of 2^{31}. Why isn't it a factor of 2^{32}?

18.6.2.(4) Suppose n is the number of some register. Under what circumstances will `DR n,n` not cause a program interruption?

18.6.3.(2)+ Write a sequence of instructions to simulate a "Divide Halfword" operation. That is, given a word dividend at **WDividen** and a halfword divisor at **HDivisor**, store the halfword quotient and remainder at **HQuotent** and **HRemaind** respectively.

18.6.4.(2)+ Suppose the dividend in a signed fixed-point division can be correctly represented in a word. Can division by a nonzero word divisor cause a fixed-point divide interruption?

18.6.5.(2) Under what circumstances can a fixed-point divide interruption occur in Figure 129 on page 276?

18.6.6.(2) Rewrite the example in Figure 130 on page 276 to round the result by adding 5 *before* dividing by 10. Determine carefully whether or not there might be a carry from the addition into the high-order register.

18.6.7.(2) Rewrite the example in Figure 131 on page 277 to round the dividend before dividing by adding or subtracting 5. Determine carefully how to handle a possible carry or borrow from the low-order to the high-order register.

18.6.8.(4) Consider the problem of simulating logical division by using arithmetic divide instructions. Sketch a code sequence that will do this.

18.6.9.(2) Suppose the SRDA instruction is not available, and you want to divide the word integer in GR1 by another in GR2. Show how you can set up the double-length dividend without multiplying by 1.

18.6.10.(2)+ Figure 131 on page 277 illustrates a rounded division with positive divisor and signed dividend. Show what changes are needed if the divisor can also be negative.

18.6.11.(3)+ Figure 130 on page 276 shows a way to compute a rounded quotient. The rounding factor 5 is half the divisor, 10. Write a sequence of instructions to generalize this by computing

```
quotient = (dividend / divisor) + 1/2
```

18.6.12.(1)+ A programmer wanted to divide the positive number in GR5 by 2, and wrote

```
SR    4,4              Clear high-order word
D     4,=F'2'          Divide c(GR5) by 2
```

Find a simpler way to do this.

18.6.13.(3)+ Write an instruction sequence showing how to calculate a rounded integer quotient using 32-bit operands, without knowing the magnitude of the divisor.

18.6.14.(2)+ A table of 15 reasonably small halfword grades is stored starting at **Grades**. Write instructions to compute their average value and store it at **AvgGrade**.

18.7. Logical (Unsigned) Division Instructions

The logical division instructions are shown in Table 89:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E397	DL	RXY	Divide Logical (32,32←32+32÷32)	B997	DLR	RRE	Divide Logical Register (32,32←32+32÷32)
E387	DLG	RXY	Divide Logical (64,64←64+64÷64)	B987	DLGR	RRE	Divide Logical Register (64,64←64+64÷64)

Table 89. Binary divide instructions

These four instructions divide a double-length unsigned dividend by a single-length unsigned divisor, giving a single-length unsigned quotient in the odd-numbered register and the unsigned single-length remainder in the even-numbered register.

If both dividend and divisor are positive, logical and arithmetic division generate the same results. For example, dividing X'00000000 FFFFFFFF' by 3 generates quotient X'55555555' and remainder 0 for both types of division.

As you might expect, negative *signed* operands can produce very different results when used as logical operands in unsigned division. For example, an arithmetic division of the maximum negative number (X'80000000') by –1 (X'FFFFFFFF') is invalid; but a logical division using the same operands gives quotient zero and remainder X'80000000' (because 2^{31} is smaller than $2^{32}-1$).

Here is a case that succeeds for arithmetic division but fails for logical division:

```
L     0,=X'80000001'     Set GR1 to -2**31+1
SRDA  0,32               Extend to 64 bits in (GR0,GR1)
D     0,=F'-1'           Arithmetic division
```

The remainder is 0 and the quotient is $+2^{31}-1$, as you would expect. For a logical division, the dividend is $(2^{64}-2^{31}+1)$ and the divisor is $2^{32}-1$, which leads to a fixed-point divide interruption because the quotient is greater than $2^{32}-1$. As another example, consider

```
L     0,=F'-2'           Set GR0 to X'FFFFFFFE'
SR    1,1                Set GR1 to X'00000000'
DL    0,=F'-1'           Divide logically by  X'FFFFFFFF'
```

Figure 136. Example of logical division

This time, both quotient and remainder are X'FFFFFFFE'!

As a final example:

```
L     0,=X'FFFFFFF8'     Initialize GR0
LR    1,0                And GR1, with the same bits
DL    0,=X'FFFFFFFF'     Divide by 2**32-1
```

The quotient is X'FFFFFFF9' and the remainder is X'FFFFFFF1'.

Exercises

18.7.1.(4) Show how you can use logical division instructions to generate the results that would be obtained by using arithmetic division instructions with the same operands.

18.7.2.(2) By evaluating the expression quotient×divisor+remainder=dividend, show that the results of the division in Figure 136 are valid.

18.8. How Division Is Done (*)

Division works much like multiplication, only in reverse. Instead of adding onto the high-order half of the accumulating product, we subtract; instead of counting down in the rightmost digit position, we count up; instead of shifting right, we shift left. As before, an example using decimal arithmetic illustrates the process.

Since we start with a dividend and divisor and wish to find a quotient and remainder that satisfy the equation

```
dividend = quotient × divisor + remainder
```

The dividend must be a double-length number.

Supposing again that our basic register length is three decimal digits, a requirement on the dividend is clear: because (a) the quotient, to fit in a register, can be at most three digits long (that is, not exceeding 999) and (b) the remainder must be less than the divisor, we must not have a dividend larger than

```
999 × divisor + (divisor-1) = 10³ × divisor - 1.
```

The factor of 10^3 is the base (10) raised to the power of the number of available digits (3). Since multiplication by 10^3 in this example is equivalent to shifting left three places, the above relation means that if the division is to produce a valid quotient, the high-order half of the dividend must be *less* than the divisor. To illustrate: if the divisor is 456, then any dividend not smaller than $456000 = 10^3 \times 456$ would produce a 4-digit quotient; if the dividend is less than or equal to $455999 = 10^3 \times 456 - 1$, the quotient can be held in three digits. Note that the three high-order digits, 455, are now less than the divisor.

Suppose we want to divide 162843 by 762. In ordinary long division, at each step we determine how many multiples of the divisor can be subtracted from the leftmost part of the dividend, and enter that number as the quotient digit. When the subtraction process has been completed, the remainder, from which no further subtractions can be made, is 537, and the quotient is 213.

```
    +   213
762)162843
    1524
     1044
      762
      2823
      2286
       537
```

Just as a check, we find that 762×213+537=162843. Using decimal registers, the division works like this:

```
  162 843       High-order part of dividend less than divisor,
  762             division may proceed.

1 628 430       Shift dividend left; save leftmost digit in an
- 762             “overflow digit” position.
                Since dividend ≥ divisor,
0 866 431       Subtract, and count up at right end.
- 762           Dividend ≥ divisor; subtract again, count up
0 104 432       Dividend < divisor, no subtraction
1 044 320       Shift dividend left again
- 762           Dividend ≥ divisor; subtract and count up
0 282 321       Dividend < divisor; no subtraction
2 823 210       Shift left for the third and last time
- 762           Dividend ≥ divisor; subtract and count up
2 061 211       Subtract and count up by 1 at right end
- 762           Dividend ≥ divisor; subtract
1 299 212         and count up by 1
- 762           Dividend ≥ divisor; subtract and count up
  537 213       Dividend now < divisor; stop.
```

As the successive digits of the quotient were developed, they appeared at the *right* end of the double-length register, and were shifted left as the division progressed. Thus at the completion of the division, the quotient is found in the *right* half of the register pair, and the remainder, from which no further subtractions could be made, is in the *left* half.

As in multiplication, binary division is simplified by the fact that at most one subtraction need be made for each quotient digit generated. To illustrate, consider an example using a five-bit divisor and a ten-bit dividend. Let the dividend be `B'00001 11011'` (=59), and let the divisor be `B'00110'` (=6). (Remember, the two halves of the double-length dividend are not two signed five-bit numbers joined end to end: the leftmost bit of the right half of the dividend is not a sign bit but an ordinary arithmetic digit.) If we make allowance for the sign bits of the quotient and remainder, we actually need an extra shift at the beginning, to align the dividend correctly. This leads to the following division scheme.

1. Shift the dividend left once. If the high-order (left) part of the dividend is not smaller than the divisor, an illegal division is being attempted.

2. Shift left one bit position. If the high-order part of the dividend is greater than or equal to the divisor, subtract the divisor from the dividend, and insert a 1 bit in the rightmost digit position. Otherwise, do nothing.
3. Return to step 2 until a total of 5 shifts has been done, including the shift of step 1.

We now illustrate the binary division of 59 by 6 in Figure 137, with less detail than in the multiplication example.

```
 00011 10110       Shift left once and compare
(00110)            Dividend < divisor, okay to continue

 00111 01100       Shift left once (second shift)
 00001 01101       Subtract divisor, insert 1
 00010 11010       Shift left once (third shift)
                   Dividend < divisor; no subtraction
 00101 10100       Shift left once (fourth shift)
                   Dividend < divisor; no subtraction
 01011 01000       Shift left once (fifth and last shift)
 00101 01001       Subtract divisor, insert 1.
```

Figure 137. Illustration of binary division

Thus the remainder `B'00101'` (=5) is in the left half, and the quotient `B'01001'` (=9) is in the right half, as expected.

This example of binary division is meant to illustrate the general process. Many improvements involving multiple dividend and divisor bits make division faster on modern processors than testing single bits.

Division in System z involves a double-length register, either a pair of general registers or an internal double-length register holding the extended single-length dividend. Since the high-order register of the pair must be even-numbered, the quotient is found in the odd-numbered register, and the remainder is found in the even-numbered register.

Exercises

18.8.1.(4) The results of a division operation must satisfy the relation

```
dividend = (quotient * divisor) + remainder.
```

However, this relation does not uniquely determine the quotient and remainder obtained from a given divisor and dividend. Even requiring the magnitude of the remainder to be smaller than the magnitude of the divisor,

```
|remainder| < |divisor|
```

does not lead to uniqueness! Consider the following choices:

1. sign(remainder) = sign(dividend) (System z)
2. remainder ≥ 0 ("modulo")
3. $-|divisor/2| \leq$ remainder $< |divisor/2|$ ("rounding")

For cases (2) and (3), show how the System z rules concerning signs and magnitudes would have to be modified.

18.8.2.(4)+ Suppose n is the number of a general register. For each of these instructions, answer the questions (1) Under what circumstances will this instruction cause an interruption? and (2) What kind or kinds of interruption?

```
1.       AR    n,n
2.       MR    n,n
3.       DR    n,n
```

18.9. Summary

Table 90 summarizes the multiply instructions we've discussed here.

Func-tion	Product length (bits)	32		32+32	64		64+64
	Operand 1 length	**32**		**32**	**64**		**64**
	Operand 2 length	**16**	**32**	**32**	**32**	**64**	**64**
Arithmetic ×		MH	MS MSR	M MR	MSGF MSGFR	MSG MSGR	
Logical ×				ML MLR			MLG MLGR

Table 90. Summary of multiply instructions discussed in this section

The divide instructions discussed in this section are shown in Table 91.

Function	Dividend length (bits)	32+32	64		64+64
	Divisor length	**32**		**64**	**64**
	Quotient & remainder length	**32**	**64**	**64**	**64**
Arithmetic ÷		D DR		DSG DSGR	
Logical ÷		DL DLR	DSGF DSGFR		DLG DLGR

Table 91. Summary of divide instructions discussed in this section

---

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
D	5D
DL	E397
DLG	E387
DLGR	B987
DLR	B997
DR	1D
DSG	E30D
DSGF	E31D

Mnemonic	Opcode
DSGFR	B91D
DSGR	B90D
M	5C
MH	4C
ML	E396
MLG	E386
MLGR	B986
MLR	B996

Mnemonic	Opcode
MR	1C
MS	71
MSG	E30C
MSGF	E31C
MSGFR	B91C
MSGR	B90C
MSR	B252

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
1C	MR
1D	DR
4C	MH
5C	M
5D	D
71	MS
B252	MSR
B90C	MSGR

Opcode	Mnemonic
B90D	DSGR
B91C	MSGFR
B91D	DSGFR
B986	MLGR
B987	DLGR
B996	MLR
B997	DLR
E30C	MSG

Opcode	Mnemonic
E30D	DSG
E31C	MSGF
E31D	DSGF
E386	MLG
E387	DLG
E396	ML
E397	DL

Terms and Definitions

arithmetic division
Division of two signed operands, generating a signed quotient and signed remainder.

arithmetic multiplication
Multiplication of two signed operands, generating a signed product.

dividend
A number to be divided by a divisor; the first operand; the numerator.

divisor
A number to be divided into the dividend; the second operand; the denominator.

logical division
Division of two unsigned operands, generating an unsigned quotient and unsigned remainder.

logical multiplication
Multiplication of two unsigned operands, generating an unsigned product.

multiplicand
In a multiplication, the number that is to be multiplied (the first operand) by another, the multiplier (the second operand)

multiplier
See *multiplicand*

quotient
The primary result of a division operation.

remainder
The residual portion of a division left over when a dividend cannot be evenly divided by a divisor. Smaller in magnitude than the divisor.

Programming Problems

Problem 18.1.(2) Write an Assembler Language program that finds the largest integer divisor x of the integer function

$$f(n) = n^3 - 1,$$

for values of n running from 2 to 8 in steps of 1, and such that “x” is less than f(n). Your program should search for the divisor, and not compute it from the known factors of f(n).

Problem 18.2.(3) Write an Assembler Language program to compute and print the values of Xn and the quotient and remainder of the fraction

$$\frac{(Xn)**2 + 10727*Xn - 14}{2*Xn - 5}$$

where Xn is given by Xn = 2**(3*n), for n = 1, 2, ..., 10.

Problem 18.3.(4) In the early 17th century, Mersenne conjectured that the number

```
M(p) = (2**p) - 1
```

is prime for a particular sequence of prime values of p. Though the conjecture is now known to be false, several efficient tests for the primality of M(p) have been devised; we will use one (due to the French mathematician Lucas) for testing a set of such "Mersenne Numbers", as follows:

1. Compute M(p), and set S(1) (the initial term of a series) to the value 4. (Note that M(p) can be calculated very simply by shifting.)
2. Compute the next term S(n+1) of the series as the remainder of the division of (S(n)*S(n) − 2) by M(p).
3. Stop when S(p−1) has been calculated, and print the values of p, M(p), and S(p−1). If S(p−1) is zero, M(p) is prime.

Write a program that tests M(p) for values of p = 3, 5, 7, 11, 13, 17, 19, 23, 29, and 31.

Problem 18.4.(3) For values of the integer variable X running from 0 to 12 in steps of 1, compute and print the quotient and remainder of the quantity

$$(X^4 + 7X^2 - 11) / (X^3 - 21X^2 + 131X - 231)$$

If you find that the denominator is zero for any value of X, print the largest negative magnitude for both quotient and remainder (that is, the word integer with hex representation `X'80000000'`).

Problem 18.5.(2) Write a program to compute a table of factorials. (Remember that we use the notation N! for the factorial of N; define 0! = 1, and N! = N*(N−1)!.) Print the values of N and N! until N! will not fit into a word; print a value of −1 for that factorial, and stop.

Problem 18.6.(4) Write a program to calculate the day and month of Easter for the year Y, using these steps:[118]

1. Divide Y by 19; keep remainder A
2. Divide Y by 100; keep quotient B and remainder C
3. Divide B by 4; keep quotient D and remainder E
4. Divide 8B+13 by 25; keep quotient G
5. Divide 19A+B-D-G+15 by 30; keep the remainder H
6. Divide A+11H by 319; keep quotient M
7. Divide C by 4; keep quotient J and remainder K
8. Divide 2E+2J-K-H+M+32 by 7; keep remainder L
9. Divide H-M+L+90 by 25; keep quotient N
10. Divide H-M+L+N+19 by 32; keep remainder P

Then, Easter Sunday is the P-th day of the N-th month of year Y. (Note that this applies to the Gregorian calendar, for years after 1582.)

Problem 18.7.(2) Write a program to print a hexadecimal addition table, like the one you created in your solution to Exercise 2.2.4.

Problem 18.8.(2) Write a program to print a hexadecimal multiplication table, like the one you created in your solution to Exercise 2.2.4.

Problem 18.9.(4) The constant "e" (2.718...) is the base of natural logarithms. Its value is defined by

```
e = Sum (k=0,∞ ) (1/k!)
```

Evaluating e by calculating the terms of this sequence is very slow (and difficult to do with fixed-point binary arithmetic, because the third and following terms are less than one). If you rewrite the value as

[118] From *Scientific American*, March 2001, page 82.

```
e-2 = (1/2)*(1+(1/3)*(1+(1/4)*(1+(1/5)*(1+(1/6)*(1+...(1/k))))...)))
```

there's an easy way to generate successive digits:

1. Multiply the rightmost (k-th) numerator term (initially 1) by 10 and divide by k.
2. Retain the remainder as the numerator for generating the next digit.
3. Multiply the next higher-order numerator by 10, add the quotient from the previous term, and divide by (k-1).
4. Repeat until k=2. At this point, the final quotient is a digit of e.
5. Repeat from the first step to generate successive digits of e.

As a general rule, the number of digits to be generated is the same as the number of terms you evaluate.

Write a program to generate the first 50 fraction digits of e, and print the value of the constant.

Problem 18.10.(2)+ Write a program that searches for and prints the 25 prime numbers less than 100.

Problem 18.11.(2) Write a program that creates a base-seven multiplication table like the one you made for Exercise 2.4.6.

19. Logical Operations

```
     11        9999999999
    111       999999999999
   1111       99        99
     11       99        99
     11       99        99
     11       999999999999
     11       999999999999
     11                 99
     11                 99
     11       99        99
1111111111    999999999999
1111111111     9999999999
```

In this section we'll examine instructions that perform logical operations, and give examples of their use. These operations are very different from logical (unsigned) arithmetic. Here, “logical” is used in the sense of the symbolic logic of truth and falsehood; the operations are often called “Boolean” operations.[119]

The basic capabilities of a computer are derived from interconnections of basic circuits performing logical functions. Some of the same logical functions are also performed by the CPU on operands in memory and in the general registers using “logical” instructions. The instructions in this section are shown in Table 92.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
44	N	RX	AND (32)	14	NR	RR	AND Register (32)
E380	NG	RXY	AND (64)	B980	NGR	RRE	AND Register (64)
46	O	RX	OR (32)	16	OR	RR	OR Register (32)
E381	OG	RXY	OR (64)	B981	OGR	RRE	OR Register (64)
57	X	RX	Exclusive OR (32)	17	XR	RR	Exclusive OR Register (32)
E382	XG	RXY	Exclusive OR (64)	B982	XGR	RRE	Exclusive OR Register (64)

Table 92. Logical operations involving general registers

There is no difference between operations involving 32- and 64-bit registers, so we'll describe only the 32-bit forms. You can easily extend the 32-bit operations to their 64-bit equivalents.

[119] George Boole (1815-1864) was a British mathematician and philosopher who wrote extensively on logic, especially in his book *An Investigation of the Laws of Thought* (1854).

19.1. Logical Operations

Unlike logical arithmetic, in which carries and borrows may propagate from a bit position to one or more of its higher-order neighbors, boolean logical operations always operate on pairs of bits, with no interactions among neighboring bits.

The three logical operations provided by System z are AND, OR, and Exclusive OR, abbreviated "XOR". These operations between *pairs* of bits produce a result depending *only* on the values of the two bits participating in the operation. The effect of the three operations is given in Figure 138. In each box, the two bits participating in the operation are given in the left column and the top row; the result bit is at the intersection of the corresponding row and column.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

Figure 138. Logical operations AND, OR, and XOR

- In the first case, the result bit is 1 only if the first *AND* the second operand bits are 1.
- In the second case, the result bit is 1 if *either* the first *OR* the second operand bit is 1.
- In the last case, the result bit is 1 if either the first *OR* second operand bits is 1, *Exclusive* of the case where both are 1 (that is, one but not both bits are 1).[120]

The AND operation is often used to set bits to zero; OR is used to set them to one; and XOR is used to change bits from zero to one and vice versa.

Sometimes the notation for logical operators is shorter, and text descriptions and formulas may use other symbols: AND is represented by "$\wedge$" (or "$\times$" or "."), OR is represented by "$\vee$" (or "+"), and XOR is represented by "$\oplus$". In high-level languages, there are many different representations for each operation. We will use the more readable forms in Figure 138.

Exercises

19.1.1.(1) Taking 1 to represent true and 0 to represent false, rewrite the three diagrams in Figure 138 as truth tables.

19.2. Register-Based Logical Instructions

In practice, the RR and RX forms of the logical operations are not used frequently. Logical operations are often used to examine and manipulate individual bits in memory, typically using the SI-type instructions that we'll see in Section 24.

For the operations in Table 93, the CC is always set.

Operation	CC setting
AND OR XOR	**0:** all result bits are zero **1:** result bits are not all zero

Table 93. CC settings by logical instructions

120 The distinction between OR and XOR often causes problems in English, where the word "or" is often interpreted one way when the other was intended. "Question: "Are you tired or hungry?" Answer: "Yes", usually implying "both".

Unlike logical arithmetic, the result of each of these logical operations is obtained by matching the corresponding bits of each operand, without interactions between neighboring bits. For example, suppose c(GR4) = X'01234567', and c(GR9) = X'EDA96521'. Then if each of the following instructions is executed, the final contents of GR4 will be as shown.

```
Operation          AND              OR              XOR
Instruction      NR    4,9      OR    4,9      XR    4,9

c(GR4)          X'01234567'    X'01234567'    X'01234567'
c(GR9)          X'EDA96521'    X'EDA96521'    X'EDA96521'
Result          X'01214521'    X'EDAB6567'    X'EC8A2046'
```

To see in more detail how these results are obtained, examine the fourth hexadecimal digit (3 and 9) for each case:

```
     AND           OR            XOR
  3  0011       3  0011       3  0011
  9  1001       9  1001       9  1001
  1  0001       B  1011       A  1010
```

Figure 139. Examples of logical operations

Exercises

19.2.1.(1) The CC settings after the logical operations indicate whether or not the result is or is not completely zero. Can you think of any reason why a CC setting to indicate a result of all 1-bits was not provided in the design of System/360?

19.3. Logical AND

The most important use of the N and NR instructions is for "masking" operations where we need to isolate or extract portions of a word. For example, suppose we want only the third of the four positive integers packed in the data word illustrated in Figure 115 on page 249. As we saw in Section 17, we can extract it by shifting in an even-odd register pair:

```
         L     0,DataWord         Get data word with integers
         SRL   0,6                Drop off fourth one
         SRDL  0,13               Move third one into GR1
         SRL   1,19               Position for storing
         ST    1,Third            Store
```

Or, we can use a only single register:

```
         L      0,DataWord        Get data word
         SLL    0,13              Drop off first and second
         SRL    0,19              Drop off fourth, and reposition
         ST     0,Third           Store
```

If the integers could have negative values, the SRL instructions would be replaced by SRA.

The following instruction sequences use Logical AND, and may be faster. (The bits of the four integers are represented by "a", "b", "c", and "d", respectively.)

```
         L     1,DataWord         B'aaaaaaaaabbbbccccccccccccdddddd'
         N     1,Mask1            B'0000000000000ccccccccccccc000000'
         SRL   1,6                B'0000000000000000000ccccccccccccc'
         ST    1,Third            Store desired third integer
         - - -
Mask1    DC    0F,BL4'1111111111111000000'  Mask: 13 0's 13 1's, 6 0's
```

The OF operand in the DC statement ensures that the bit pattern at **Mask1** falls on a word boundary; type B constants have no implied alignment, and are padded on the left with zero bits.

We can do the same extraction by shifting first and then ANDing:

```
         L     1,DataWord         B'aaaaaaaaabbbbccccccccccccdddddd'
         SRL   1,6                B'000000aaaaaaaaabbbbcccccccccccc'
         N     1,MASK2            B'0000000000000000000cccccccccccc'
         ST    1,Third            Store Result
         - - -
Mask2    DC    A(X'1FFF')         13 1-bits at right end of word
```

Both masks have 1-bits only in positions corresponding to the bits of the third integer of the data word (named "c"). When the N instruction is executed, all of the bit positions where a mask bit is zero are set to zero, since a 0-bit ANDed to any other bit gives a zero result. In all of the mask's 1-bit positions, the result is the same as the original bit from the data word, because a 1-bit ANDed to any other bit gives a result identical to the other bit, as we saw in Figure 138 on page 289.

Exercises

19.3.1.(1)+ In the second example in Section 15.2 on page 205, shifts were used to set the leftmost 7 bits of GR8 to zero. Show how to do this with a logical AND operation.

19.4. Logical OR

In Figure 115 on page 249, we wanted to insert a new value for the third integer into the proper part of the data word. We could do this by shifting the various pieces into place:

```
         L     0,DataWord         Get 4 packed integers
         SRDL  0,6                Move fourth into GR1
         L     0,NewThird         Get new value of third integer
         SRDL  0,13               Move it in with fourth
         L     0,DataWord         Get integers again
         SRL   0,19               Drop old third and fourth
         SRDL  0,13               Move full word into GR1
         ST    1,DataWord         Store updated result
```

Using the AND and OR instructions, we can use logical operations:

```
         L     0,DataWord         Get 4 packed integers
         N     0,MaskC            Clear a space for third (C's)
         L     1,NewThird         Get new value of third integer
         SLL   1,6                Shift into proper position
         OR    0,1                'OR' into place in GR0
         ST    0,DataWord         Store new dataword
         - - -
         DS    0F                 Align
MaskC    DC    X'FFF8003F'        13 0-bits in third-integer position
```

Figure 140. Inserting a new integer value using AND and OR

The N instruction zeros all the bit positions into which the third integer will be placed. The OR instruction then forms the logical OR of all the bits of GR0 and GR1. Since the only bits in GR1 that might be ones are in the 13 positions corresponding to the space provided in the word in GR0, and because the result of ORing a zero bit to any other bit is the value of the other bit, the effect is to insert the new value of the third integer in its proper position in GR0. This of course assumes that the contents of **NewThird** is a positive integer of at most 13 significant bits; if not, an

```
         N     1,Mask1
```

instruction should be inserted before the OR instruction to ensure that no extraneous bits are ORed into GR0.

Exercises

19.4.1.(2)+ The word at **Data** contains information to be shifted *circularly*: that is, bits shifted off one end of the register should reappear at the other end. For example, a circular left shift of the operand X'12345678' by 12 bit positions would produce X'45678123'. Without using a rotating shift, write a code sequence using logical operations to shift c(Data) circularly to the left by N places, where N is a nonnegative word integer stored at **NShifts**. Compare your solution to the solution you found for Exercise 17.3.17.

19.4.2.(2)+ Modify the coding of exercise 19.4.1 so that if N is negative, the shift is a circular *right* shift instead. Again, don't use a rotating shift. Compare your solution to the solution you found for Exercise 17.3.18.

19.4.3.(2)+ What will happen if the instructions OR 3,3 and NR 3,3 are executed? what is the difference between these two and LTR 3,3 ?

19.4.4.(2) Write a code sequence using logical instructions to unpack each of the four integers illustrated in Figure 115 on page 249.

19.4.5.(2) Now that you have completed Exercise 19.4.4, rewrite your solution to Exercise 17.3.10 to pack the four integers into the word illustrated in Figure 115 on page 249, but now use logical instructions.

19.5. Logical Exclusive OR

The X and XR instructions are used to invert bits. We saw in Figure 138 on page 289 that the effect of XORing a 0-bit to any other bit is to leave it undisturbed, and the effect of XORing a 1-bit is to invert it from 1 to 0 or from 0 to 1. Any bit XORed with itself gives a zero bit. This gives a simple way to set a register to zero.[121]

```
         XR    1,1                 Set GR1 to zero
```

We can rewrite Figure 140 on page 291 (in a somewhat roundabout way) to use an X instruction:

```
         L     0,DataWord          Get integers
         O     0,Mask3             Set third-integer space to all 1's
         X     0,Mask3             Now set them to zeros
         L     1,NewThird          Etc., as before
         SLL   1,6                 Etc.
         N     1,Mask3             Make sure there are no extra bits
         OR    0,1                 Etc.
         ST    0,DataWord          Store updated result
         - - -
         DS    0F
Mask3    DC    X'0007FFC0'
```

Figure 141. Data masking using Exclusive OR

The O instruction first sets all bits in the third integer's position to 1-bits, and the X instruction then resets them all to zero. We'll see another use of this technique in Figure 143 on page 293.

As another example of the use of the Exclusive OR instruction, suppose we want to force the integer in GR7 to be the next larger multiple of 8 if it is not already a multiple of 8. (We saw a different way to do this in Figure 109 on page 246.) Consider the two following code segments.

[121] This is a very efficient way to zero a general register, because (unlike subtracting the register's contents from itself), the CPU need not check for a possible overflow.

```
         A     7,=F'7'              Force carry if any 1s in low 3 bits
         N     7,=F'-8'             Now, set last 3 bits to zero
```

Figure 142. Rounding to the next multiple of 8

That is a faster method, but space is required for the two constants. We can also use the “OR then XOR” technique:

```
         LH    0,=H'7'              c(GR0) = 7 = alignment mask
         AR    7,0                  Force carry if any 1's in low 3 bits
         OR    7,0                  Now force those three bits to 1
         XR    7,0                  And now set them to zero
```

Figure 143. Rounding to the next multiple of 8

This method is more economical of total instruction length than those illustrated previously.

As a more detailed example, suppose we need to shift the (nonzero) integer contents of GR6 to the left so that the most significant bit is immediately to the right of the sign bit, and store the number of positions shifted at **Norm**. The most significant bit is the leftmost bit that differs from the sign bit.

```
         XR    8,8                  Set shift count in GR8 to zero
Shift    SLA   6,1                  Shift left one bit position
         BO    Finish               Branch if overflowed
         AH    8,=H'1'              Increment shift count
         B     Shift                Try again
Finish   SRA   6,1                  Reposition
         X     6,Digit              Restore the lost bit
         ST    8,Norm               Store shift count
         - - -
Norm     DS    F                    Storage space and alignment
Digit    DC    X'40000000'          Mask bit for lost bit
```

We shift left until the overflow condition indicates that a bit different from the sign bit has been shifted out of bit position 1. The following right shift moves everything back in place, but instead of restoring the lost bit, extends the sign bit into the second bit position of R6, from which the most significant bit was just lost. Since the sign is known to be the opposite of the lost bit, the X operation inverts the second bit to give the correct result.

We can form the ones' complement of the number in GR7 by subtracting it from a word of all 1-bits, or by executing

```
         X     7,=F'-1'
```

that does the same thing more simply. Thus, we can use the X instruction to form the two's complement of a double-length integer, as in Figures 90 and 91 on page 226.

```
         LM    8,9,Arg              64-bit operand in (GR8,GR9)
         X     8,=F'-1'             Ones' complement of high-order part
         X     9,=F'-1'             Ones' complement of low-order part
         AL    9,=F'1'              Add low-order 1-bit
         BC    B'1100',NoCarry      Branch if no carry out
         AL    8,=F'1'              Add carry into high-order part
NoCarry  STM   8,9,ARG              Store complemented result
         - - -
Arg      DS    2F                   Double-length word integers
```

Figure 144. Complementing a double-length integer

This is definitely not the most efficient way to form a complement, but does show one use of XOR.

Exercises

19.5.1.(2)+ Show by examining the possible bit patterns that the sequence of instructions given below exchanges the contents of GR1 and GR2 without using any other register.

```
         XR    1,2
         XR    2,1
         XR    1,2
```

Can the same be done between a register and a word in memory, using three instructions?

19.5.2.(2) What is the result of replacing the XR instructions in Exercise 19.5.1 with SR instructions?

19.5.3.(4) Suppose you are programming on a processor that has addition and subtraction operations, a logical AND operation, but no OR or Exclusive OR.[122] By examining various bit combinations (particularly at the left end of a register), show that you can compute the missing logical functions from

```
   A  OR  B  =   (A + B) - (A AND B)
   X XOR  B  =  (A OR B) - (A AND B)
```

19.5.4.(2)+ Consider these four logical expressions:

```
  (1)  A XOR  (A XOR B)
  (2)  A XOR  (B XOR A)
  (3) (A XOR B)  XOR A
  (4) (B XOR A)  XOR A
```

What is the result of each operation?

19.5.5.(2)+ Figure 141 on page 292 was rewritten by a student as follows:

```
         L     0,DataWord        Get old packed integers in GR0
         L     1,NewThird        Get new third integer in GR1
         SLL   1,6               Position new value correctly
         XR    1,0               XOR with old data in GR0
         N     1,Mask3           Mask all but 3rd integer's GR1 bits
         XR    0,1               XOR those bits back into GR0
         ST    0,DataWord        Store updated packed result
```

where **Mask3** defines the same bit pattern. By suitable examples, prove that this program segment either does or does not work.

19.5.6.(2) Rewrite Figure 142 on page 293 to use a single literal. Are any new problems created in testing the Condition Code?

19.5.7.(2) Write a DC statement with an A-type constant to specify the mask in Figure 141 on page 292.

19.5.8.(2) Write code sequences using logical instructions to extract the first, second, and fourth integers packed in a word at **DataWord** in the format illustrated in Figure 115 on page 249, and store the resulting values in the words at **First**, **Second**, and **Fourth**.

19.5.9.(3) The word at **Pack** contains four positive integers in the format illustrated in Figure 115 on page 249. Write a code sequence that will retrieve and store at **DataItem** the first, second, third, or fourth of the packed binary integers, depending on the value of the halfword binary integer stored at **ItemNbr**, which may have value 1, 2, 3, or 4. (It may help to use tables of masks and shift counts.)

---

[122] This was true of some very early "Von Neumann" or "Institute-type" processors like the ILLIAC 1.

19.6. Interesting Uses of Logical Instructions (*)

The examples of logical instructions in the previous sections show “normal” uses. You can do some other interesting things with them; we will illustrate a few.[123]

1. Test a nonzero, nonnegative number to see if it's a power of 2:

```
Y = ((2*X)-1) AND X) XOR X
```

If Y is zero, X is a power of 2. (Note that if X is zero or is the maximum negative number, Y=0.) To illustrate:

```
L     0,=F'5'          X in GR0                X'00000005'
LR    1,0              Copy X to GR1           X'00000005'
SLL   1,1              2*X                     X'0000000A'
S     1,=F'1'          (2*X-1)                 X'00000009'
NR    1,0              (2*X-1) AND X           X'00000001'
XR    1,0              ((2*X-1) AND X) XOR X   X'00000004'
JZ    PowerOf2         Branch if a power of 2
```

so that 5 is not a power of 2.

2. Isolate a number's rightmost 1-bit. If X is a nonzero, nonnegative number:

```
Y = (((X-1) XOR X)+1)/2
```

then Y is the rightmost 1-bit of X. To illustrate:

```
L     0,=F'6'          X in GR0                X'00000006'
LR    1,0              Copy X to GR1           X'00000006'
S     1,=F'1'          (X-1)                   X'00000005'
XR    1,0              (X-1) XOR X             X'00000003'
A     1,=F'1'          ((X-1) XOR X)+1         X'00000004'
SRL   1,1              Y=(((X-1) XOR X)+1)/2   X'00000002'
```

which is the rightmost bit of 6 = B'00...0110'. If X is zero or the maximum negative number, Y will be zero.

3. Turn off the rightmost 1-bit of a positive binary number X:

```
Y = X AND (X-1)
```

To illustrate:

```
L     0,=F'6'          X in GR0                X'00000006'
LR    1,0              Copy X to GR1           X'00000006'
S     1,=F'1'          (X-1)                   X'00000005'
NR    1,0              (X-1) AND X             X'00000004'
```

If this process is repeated, the number of iterations is determined by the power of two represented by the leftmost 1-bit.

4. Right-propagate the rightmost 1-bit of a nonzero word:

```
Y = X OR (X-1)
```

To illustrate:

```
L     0,=F'12'         X in GR0                X'0000000C'
LR    1,0              Copy X to GR1           X'0000000C'
S     1,=F'1'          (X-1)                   X'0000000B'
OR    1,0              (X-1) OR X              X'0000000F'
```

5. Isolate the rightmost 1-bit of a word:

```
Y = X AND (-X)
```

To illustrate:

---

[123] Some of these examples are based on IBM Thomas J. Watson Research Center Report RC 5809 *Functions Realizable with Word-Parallel Logical and 2's-Complement Addition Instructions* by Henry S. Warren, Jr.

```
         L     0,=F'12'          X in GR0               X'0000000C'
         LCR   1,0               Copy -X to GR1         X'FFFFFFF4'
         NR    1,0               (-X) AND X             X'00000004'
```

6. Turn off the rightmost contiguous string of 1-bits in a word:

   `Y = [(X OR (X-1)) + 1] AND X`

   To illustrate:

```
         L     0,=F'23'          X in GR0               X'00000017'
         LR    1,0               Copy X to GR1          X'00000017'
         S     1,=F'1'           (X-1)                  X'00000016'
         OR    1,0               (X-1) OR X             X'00000017'
         A     1,=F'1'           ((X-1) OR X)+1         X'00000018'
         NR    1,0               (((X-1) OR X)+1) AND X X'00000010'
```

   and B'00..010111' becomes B'00..010000'.

7. Left-propagate the bit at position *k* in a word:

   Y = [(X AND (2^{k+1})) XOR 2^k] − 2^k

   A more "natural" way to program this might be to write:

```
         L    0,X
         SLL  0,K
         SRA  0,K
```

   but that wouldn't be as interesting.

8. Test if a number is a power of 2, minus 1:

   `Y = (X XOR (X+1))`

   if Y is zero, X is of the form $2^N - 1$.

   To illustrate:

```
         L     0,=F'31'          X in GR0               X'0000001F'
         LR    1,0               Copy X to GR1          X'0000001F'
         A     1,=F'1'           (X+1)                  X'00000020'
         XR    1,0               (X+1) XOR X            X'00000000'
```

   so 31 is a power of 2 minus 1.

Exercises

19.6.1.(2)+ In example 1 of this section, it is stated that if X is 0 or the maximum negative number, Y=0. Verify this statement.

19.6.2.(3) In example 1 of this section, what result Y is obtained if X is the negative of a number that is a power of 2?

19.6.3.(3) In example 2 of this section, what result Y is obtained if X is zero? What result is obtained if X is a negative number?

19.6.4.(2)+ In example 3 of this section, what will happen if X is a negative number?

19.6.5.(2)+ In example 3 of this section, what will happen if X is zero?

19.6.6.(2)+ In example 4 of this section, what will happen if X is zero? If X is negative?

19.6.7.(2) In example 5 of this section, what will happen if X is zero? If X is negative?

19.6.8.(2)+ In example 6 of this section, what will happen if X is zero? If X is negative?

19.6.9.(2) Example 7 above shows how to left-propagate a bit in a general register. Suppose there is an integer K between 1 and 31 stored in the word at **KWord**. Write a code sequence that

will left-propagate the bit in position K of the word in GR5, using the detailed formula (not the "natural" solution).

19.6.10.(2)+ Use the techique of example 3 of this section to count the number of 1-bits in the word in GR0, and leave the result in GR2.

19.6.11.(2) In example 8 of this section, will the technique work if the value of X is unsigned?

19.6.12.(3)+ It is claimed that this formula:

```
(NOT X) AND (X+1)
```

will create a mask that isolates the rightmost zero bit of X. That is, if X=7, the resulting mask is `X'00000008'`. Write instructions testing a range of negative and positive values of X to validate or invalidate this claim. What will be the result if X=0?

19.6.13.(3)+ It is claimed that all three of these formulas:

```
(NOT X) AND (X-1),  NOT(X OR -X),  and  (X AND -X)-1
```

will form a mask matching all trailing zero bits. That is, if X=12, the resulting mask is `X'00000003'`. Write instructions testing a range of negative and positive values of X to validate or invalidate this claim for all three formulas. What will be the result if X=0?

19.6.14.(3)+ It is claimed that this formula:

```
X XOR (X-1)
```

will form a mask matching the rightmost one bit of X, and all trailing zero bits. That is, if X=8, the resulting mask is `X'0000000F'`. Write instructions testing a range of negative and positive values of X to validate or invalidate this claim. What will be the result if X=0?

19.7. Summary

Table 94 gives a compact summary[124] of the three logical operations:

Operation	**Anything with**		
	One	**Zero**	**Itself**
AND	It remains unchanged	It is changed to zero	It remains unchanged
OR	It is changed to one	It remains unchanged	It remains unchanged
XOR	It is inverted	It remains unchanged	It is changed to zero

Table 94. Summary of the logical operations AND, OR, XOR

The instructions discussed in this section are summarized in Table 95.

Function / **Operand length (bits)**	**32**	**64**
AND (memory)	N	NG
AND (register)	NR	NGR
OR (memory)	O	OG
OR (register)	OR	OGR
XOR (memory)	X	XG
XOR (register)	XR	XGR

Table 95. Logical-operation instructions discussed in this section

[124] Courtesy of Michael Stack.

Exercises

19.7.1.(4) Given the four logical operations AND, OR, XOR, and NOT, where (NOT A) is equivalent to (1 XOR A): which of each can be expressed in terms of two of the other three?

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
N	54
NG	E380
NGR	B980
NR	14

Mnemonic	Opcode
O	56
OG	E381
OGR	B981
OR	16

Mnemonic	Opcode
X	57
XG	E382
XGR	B982
XR	17

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
14	NR
16	OR
17	XR
54	N

Opcode	Mnemonic
56	O
57	X
B980	NGR
B981	OGR

Opcode	Mnemonic
B982	XGR
E380	NG
E381	OG
E382	XG

Terms and Definitions

AND operation
A logical (boolean) operation between two bits, whose result is 1 only if both operand bits are 1.

OR operation
A logical (boolean) operation between two bits, whose result is 1 if either operand bit is 1.

XOR operation
A logical (boolean) operation between two bits, whose result is 1 if either operand bit is 1 while the other is zero. If the operand bits are identical, the result is zero.

Programming Problems

Problem 19.1.(4) In binary addition, the sum S of two binary digits A and B is

```
S = A XOR B,
```

and the carry bit is

```
c = A AND B.
```

Thus, to add two numbers composed of a string of binary digits, we must form the sum bit S(i) of the appropriate digits A(i) and B(i), as well as the carry bit from the next lower-order digit position, c(i−1). The logical formulas for the sum and carry digits then become

```
S(i) =  A(i) XOR B(i) XOR c(i-1)
```

and the new carry bit is

```
c(i) = (A(i) AND B(i)) OR (B(i) AND c(i-1)) OR (A(i) AND c(i-1))
```

That is, c(i) is 1 if two or more of A(i), B(i), and c(i−1) are 1.

Write a program that computes the logical sum of several pairs of words A and B by performing the above operations 32 times, once on each bit position in the word in succession. Save or calculate enough information during this process so that when the operation is complete, you can store a byte at **CCL** whose value is the same as the CC setting that would result if the AL or ALR instructions had been used to add the same operands. Your sample values should generate all four possible CC values.

If you can, store at **CCA** a byte whose value is the same as the CC setting that would result if the A or AR instructions had been used to add the same operands.

Thus, you should detect the presence or absence of a final carry, and whether the result is zero or nonzero and positive or negative, by examining the bits as the operation progresses.

Problem 19.2.(3) Write a code sequence that forms the logical sum of two word operands A and B, using the same logical formulas as in Problem 19.1. In this case, however, the operations should be performed on all 32 bits at once. (Show that there is no interference between neighboring bit positions.) One method is to generate a word containing

```
S(1) = A XOR B
```

and a word

```
c(1) = A AND B.
```

The word S(1) contains the sum digits for the first addition, and the word c(1) contains the carries generated in the first addition step. Shift c(1) left one bit position, and repeat the cycle by ANDing and XORing S to c, generating a new sum S(2) and a new set of carries c(2). Repeat the process until either c(n) is zero for some n, or 32 steps have been done. That is,

```
S(n+1) = S(n) XOR (2*c(n))
c(n+1) = S(n) AND (2*c(n))
```

Store the final sum at **Sum**, and set the word at **CCodeL** to contain the value of the Condition Code setting as it would have been produced by the AL or ALR instructions.

Problem 19.3.(2) Modify the logical operation sequences in Problem 19.2 (or in Problem 19.1) to perform additions *or* subtractions, as indicated by whether the word at **SubFlag** is or is not zero. Test your program on a representative set of values for A and B.

Problem 19.4.(3) There are two parts to this problem. First, a small table of prime numbers is computed using a method called the "Sieve of Eratosthenes", and then the table is condensed for printing.

To construct the table of primes, lay out in memory a table area of 400 units of any convenient size; the choice of size is up to you. Consider them to be numbered from 1 to 400. Then, beginning with table entry number 2, mark in some way each multiple of 2 (other than 2 itself), up to 400. Then find the next unmarked quantity in the table (which will be 3), and mark each multiple of that number. Then search for the next unmarked number (which will be 5), and continue in this fashion.

Only prime numbers will remain unmarked. You need not make passes over the table marking multiples of any number greater than 19, since the first unmarked number to be marked in this "sieving" process will be the square of the number whose multiples are being marked.

From this table, produce a condensed version in a string of 400 *bits* (50 bytes) such that 1-bits indicate that the corresponding number is unmarked (and therefore prime). Define the string in a statement such as

```
PrimeBts DC    XL50'00'           Space for 400 bits
```

so that an appropriate single statement will print the entire string of 100 hexadecimal digits, that should start with X'EA28...', representing 1, 2, 3, 5, 7, 11, 13, ...

If you wish, you may compute the final bit string directly, without having to go through the intermediate steps of forming a byte table.

Problem 19.5.(3) In Problem 19.4, you produced a string of bits indicating whether the number that gave its position in the string was a prime number. Half the bits in the table are wasted, since all even numbers except 2 cannot be prime.

Write a program that will produce a string of 200 bits (25 bytes) indicating which of the *odd* numbers less than 400 are prime. That is, if the k-th bit of the string is a 1-bit, the number 2k-1 is prime. Your string of 200 bits should start with `X'F6D32D...'`, representing 1, 3, 5, 7, 11, 13, ...

If you had a string of 2^{30} bytes (1GB) available for storing the bits, what is the largest prime whose primality you could indicate in that bit string?

Problem 19.6.(2) Choose an example from Section 19.6 on page 295 and write a program to test the given formula for a range of values.

Chapter VI: Addressing, Immediate Operands, and Loops

```
VV        VV  IIIIIIIIII
VV        VV  IIIIIIIIII
VV        VV      II
VV        VV      II
VV        VV      II
VV        VV      II
VV        VV      II
 VV      VV       II
  VV    VV        II
   VV  VV         II
    VVVV      IIIIIIIIII
     VV       IIIIIIIIII
```

The previous chapters have described many different types of instructions. Recent additions to the original System/360 architecture include extensions to those basic types that can make your programs more efficient, and often much easier to write.

- Section 20 describes different types of address generation and the important concept of addressing modes and the very useful "Load Address" instruction.
- Section 21 introduces instructions with *immediate* operands that operate on data in the general registers.
- Section 22 examines old and new forms of branch instructions, some of which have immediate operands. These instructions help manage loops efficiently for iterative processing.

20. Address Generation and Addressing Modes

```
 2222222222      00000000
222222222222    0000000000
22        22   00        00
          22   00        00
          22   00        00
         22    00        00
       22      00        00
     22        00        00
   22          00        00
 22            00        00
222222222222    0000000000
222222222222     00000000
```

20.1. Address Generation

System z provides three forms of Effective Address generation:

1. base-displacement with unsigned 12-bit displacements;
2. base-displacement with signed 20-bit displacements; and
3. relative-immediate.

The next three subsections will describe them.

20.1.1. Address Generation With 12-Bit Displacements

We saw in Sections 5.1 and 5.3 on pages 62 and 63 how Effective Addresses are generated from instructions using base-displacement addressing: the CPU adds the displacement to the contents of the base register (and the index register, if any is specified). Figure 19 on page 62 and Figure 21 on page 64 illustrate the process.

In this form, 12-bit displacements are limited to the range

$$0 \leq \text{displacement} \leq +2^{12}-1, \quad \text{or} \quad 0 \leq \text{displacement} \leq +4095.$$

20.1.2. Address Generation With 20-Bit Displacements

In Section 14.7 we saw examples of RXY-type instructions (like LG and STG) that use a 20-bit signed displacement. Table 96 illustrates the RXY- and RSY-type instruction formats:

opcode	R_1	X_2	B_2	DL_2	DH_2	opcode

Table 96. Format of RXY- and RSY-type instructions

For RSY-type instructions the X_2 field is replaced by an R_3 field, but that doesn't affect address generation other than not supporting indexing.

An Effective Address is generated for these "long-displacement" instructions in much the same way it is generated for RX and similar types with an unsigned 12-bit displacement. In this case the displacement is a *signed* 20-bit number; the displacement fields are rearranged and combined as shown in Figure 145.

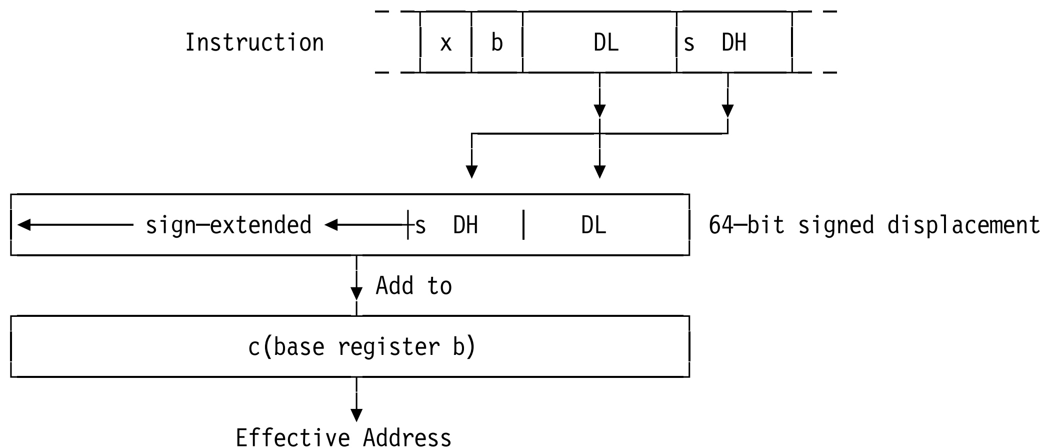


Figure 145. Effective Address generation for long-displacement instructions

In these instructions, the traditional 12-bit unsigned displacement field (named "D") is now named "DL", and the high-order 8-bit *signed* displacement extension is named "DH". A 20-bit signed displacement is formed from DH and DL: DH is concatenated at the left end of DL, and then sign-extended to 64 bits. This gives a displacement value in the range

$$-2^{19} \le \text{ displacement } \le +2^{19}-1, \quad \text{or} \quad -524288 \le \text{displacement} \le +524287.$$

rather than the limited 12-bit displacement range

$$0 \le \text{ displacement } \le 4095.$$

If the DH field is zero, the result is generated from the familiar 12-bit unsigned displacement.

If the instruction is RXY-type, the address calculation adds both the base and index register contents, if applicable.

The Assembler uses the same resolution rules described in Sections 10.9 (on page 127) and 10.13 (on page 132) with one added step:

5. If no nonnegative displacement can be assigned, choose the register giving a negative displacement with the smallest magnitude.

To illustrate, suppose **X** has value X'2468A0'. With traditional 16-bit addressing halfwords, these statements would fail:

```
Using  X,3
L      9,X-4              Addressability error
```

The operand X-4 is not addressable, because the RX-type instruction L provides only an unsigned 12-bit displacement. The LY instruction has an extended 3-byte base-displacement, so that

```
Using  X,3
LY     9,X-4
```

will resolve the implied address with an extended 3-byte base-displacement X'3 FFC FF', where the "true" displacement from the base location in GR3 to the operand location is X'FFFFC'. That is, B_2 = 3, DL_2 = X'FFC', and DH_2 = X'FF'.

The instructions

```
Using  X,3
LY     9,X+4
```

will resolve the implied address with an extended 3-byte base-displacement X'300400', where the traditional 16-bit addressing halfword X'3004' is in the first two bytes and DH_2 = X'00'.

Long displacements provide far greater addressability than the traditional 12-bit displacements, which are limited to 4KB.

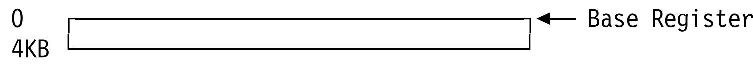


Figure 146. Addressability range with 12-bit displacements

You can address very large data areas with a *single* base register, by setting the base address at (or near) the "middle" of the area, as shown in Figure 147.

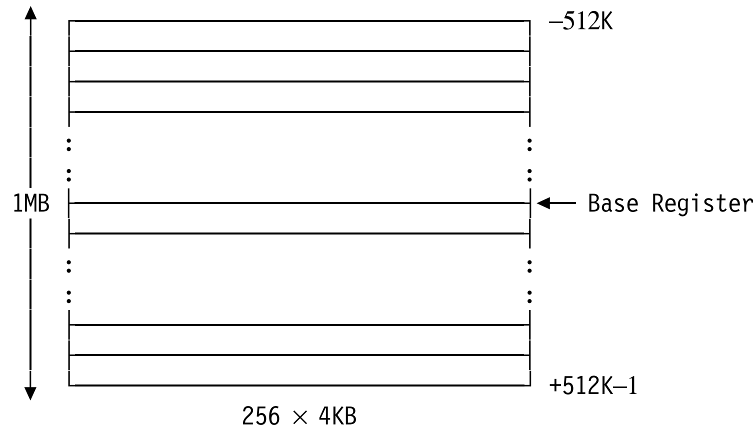


Figure 147. Addressability range with 20-bit displacements

With 12-bit unsigned displacements, addressing 1MB could require 256 base registers.

Some RX-type and SI-type instructions have equivalent forms with long displacements. They are shown in the following table (and some of them will be described later).

12-bit displacement	20-bit displacement
A	AY
AH	AHY
AL	ALY
C	CY
CH	CHY
CL	CLY
CLI	CLIY
CLM	CLMY
CVB	CVBY
CVD	CVDY
IC	IC
ICM	ICMY
L	LY

12-bit displacement	20-bit displacement
LA	LAY
LD	LDY
LE	LEY
LH	LHY
LM	LMY
M	MY
MH	MHY
MS	MSY
MVI	MVIY
N	NY
NI	NIY
O	OY
OI	OIY

12-bit displacement	20-bit displacement
S	SY
SH	SHY
SL	SLY
STCM	STCMY
STC	STCY
STD	STDY
STE	STEY
STH	STHY
STM	STMY
ST	STY
TM	TMY
X	XY
XI	XIY

There are many other instructions with long displacements that are not direct extensions of other RX-type and SI-type instructions.

20.1.3. Address Generation With Relative-Immediate Operands

The formats of the two relative-immediate instruction types are shown in Tables 97 and 98.

Opcode	R_1	Op	RI_2

Table 97. Format of R-I instructions with 16-bit immediate operands

Opcode	R_1	Op	RI_2

Table 98. Format of R-I instructions with 32-bit immediate operands

Unlike the arithmetic and logical immediate operands we'll see in Sections 21.1 through 21.3, these RI_2 *relative-immediate* operands do not involve data in memory or in a general register. Instead, they are used to form the Effective Address:

1. Sign-extend the immediate operand to 64 bits, and shift it left once, giving $2 \times RI_2$.
2. Add the address of the *current* relative-immediate instruction (*not* the address in the IA of the PSW); the result is the Effective Address. Thus, the Effective Address is *relative* to the address of the current instruction.

This process is illustrated in Figure 148.

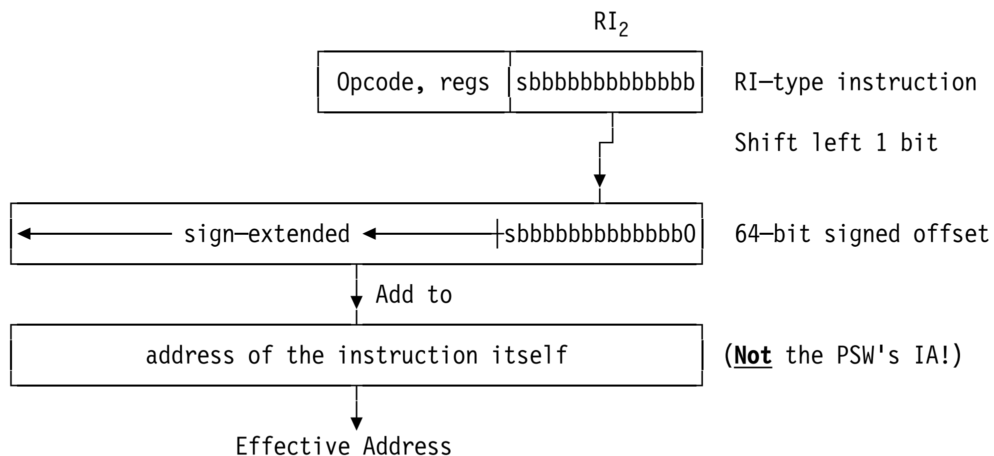


Figure 148. Effective Address formation for relative-immediate instructions

In effect, you have added or subtracted the number of *halfwords* specified by the RI_2 operand to the address of the instruction.[125] The signed RI_2 value means that the Effective Address can either precede or follow the address of the instruction. For 16-bit RI_2 fields,

$$\text{Instruction's address} - 65536 \leq \text{Effective Address} \leq \text{Instruction's address} + 65534,$$

and for 32-bit RI_2 fields,

$$\text{Instruction's address} - 4294967296 \leq \text{Effective Address} \leq \text{Instruction's address} + 4294967294.$$

Both these “offsets” from the instruction's address are adequate for most programs.

To resolve the implied addresses of instructions with relative addressing, the Assembler calculates the difference between the locations of the operand and the instruction and divides the result by 2. The target operand must

[125] The RI_2 operand is doubled because the Effective Address usually forms a branch address, which must always refer to a halfword boundary. For some other processor architectures, the Instruction Address is called the “Program Counter”, and Effective Addresses calculated relative to the address of the instruction are then called “PC-relative”.

- be aligned on a halfword boundary, and
- have the same relocation attribute as the instruction. (This rule can be relaxed if the target operand is an external symbol, as we'll see in Section 38.)

For example, a “Branch Relative on Condition” instruction (we'll discuss it in Section 22.1) might look like this:

```
         BRC   8,Target           Branch if Condition Code 0
         - - -
Target   L     0,NewValue
```

and the Assembler will calculate the correct RI_2 offset from the BRC instruction to the Target instruction.

Exercises

20.1.1.(1) The RI-type instruction at address X'174629C' generates an Effective Address. For each of the four following RI_2 operands, show the generated Effective Address. Assume the generated address is 32 bits long.

1. −1
2. 6845
3. −65536
4. 2

20.1.2.(1) The RIL-type instruction at address X'7B1EF0' generates an Effective Address. For each of the following RI_2 operands, show the generated Effective Address. Assume the generated address is 32 bits long.

1. −1
2. 384593
3. −512044
4. 3

20.1.3.(2) Suppose c(GR4)=X'FFFFFF7C' and c(GR7)=X'9610B6C0'. Show the Effective Address generated by each of these instructions. Assume the generated address is 32 bits long.

1. `L    0,0(4,7)`
2. `L    0,3624(4,7)`
3. `LG   0,4(4,7)`
4. `LG   0,-8194(4,7)`

20.1.4.(1) In Figure 148 on page 305, there is a comment saying “(Not the PSW's IA!)”. Why?

20.1.5.(1) Relative address offsets can be either 2 or 4 bytes long. What is the maximum allowed distance to an operand from a referencing instruction with (a) a 2-byte offset, (b) a 4-byte offset?

20.1.6.(1) Suppose a relative-immediate instruction is at address X'27B9AE'. For each of the following four 2-byte immediate operands, what is the Effective Address of the instruction?

(1) X'0003'
(2) X'FFE4'
(3) X'700F'
(4) X'8000'

20.1.7.(2) How can you generate an odd Effective Address using relative-immediate operands?

20.1.8.(1) Some coders refer to operands like “A+8” and “*+6” as “relative addressing”. How would you describe such operands?

20.2. Addressing Modes

We've seen how an Effective Address is generated; what happens when we use it? The answer depends on the CPU's current *addressing mode*, often abbreviated “AMode”. All the instructions we've discussed have ignored AMode considerations; we now consider some basic aspects of this important topic.

System z supports three addressing modes: 24-bit, 31-bit, and 64-bit. 24-bit addressing was used in the original System/360, when memory was very expensive: a large processor may have had as much as 256K bytes of storage, and many had far less.[126] 24-bit addresses could reference up to 2^{24} (16 million) bytes, which seemed so large that 24-bit Effective Addresses were expected to be enough for a very long time. Continued application growth was managed by adding virtual addressing facilities in the early 1970s, but addresses were still limited to 24 bits.[127]

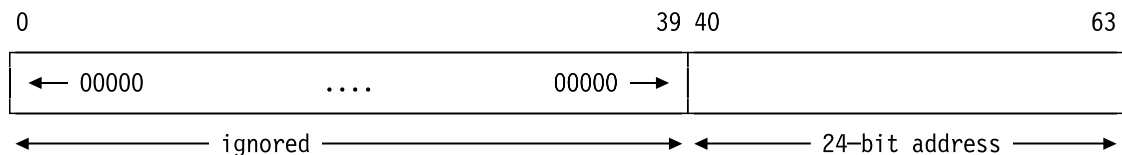
In the late 1970s and early 1980s, rapid application growth required more addressability; 31-bit addressing was introduced, which provided addressability up to 2G bytes. Because existing applications usually needed to continue executing using 24-bit addressing, great care was taken to ensure that addressing extensions were compatible with older applications.

The growth demands on applications and operating systems continued. Techniques like partitioning[128] allowed some relief, but it was soon clear that more than 31-bit addressing was needed, at least to manage physical memories much larger than 2G. Thus, in the early 2000s, 64-bit addressing and 64-bit general registers were introduced with z/Architecture.

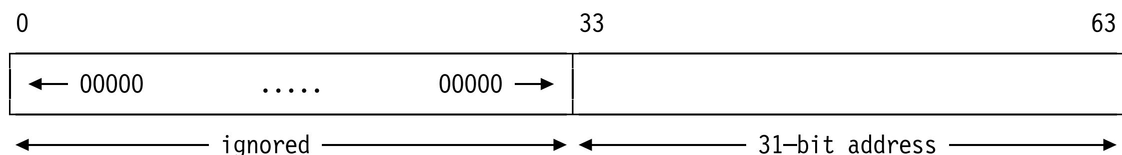
When 31-bit addressing was introduced, it was necessary to distinguish areas of memory addressable with 24-bit Effective Addresses — that is, addresses between 0 and $2^{24}-1$ — from addresses requiring 31-bit Effective Addresses. The separation between these areas was called the “line”, so that the first 2^{24} bytes were “below the line” and the rest were “above the line”. Similarly, when 64-bit Effective Addresses were provided with System z, the separation of areas having addresses less than 2^{31} and those having larger addresses was called the “bar”, so that bytes having addresses between 0 and $2^{31}-1$ were “below the bar” and those with greater addresses were “above the bar”.

Each of the three addressing modes affects the *generation* of z/Architecture Effective Addresses:

- in 24-bit mode, the leftmost 40 bits of the Effective Address (0-39) are set to zero, leaving the rightmost 24 bits intact.



- in 31-bit mode, the leftmost 33 bits of the Effective Address (0-32) are set to zero, leaving the rightmost 31 bits intact.

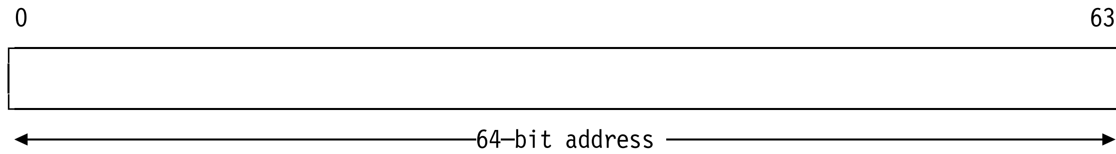


[126] Some of the most popular System/360 models had only 32K bytes of storage, of which 14K was needed for the operating system, leaving 18K bytes for applications. Programs were written *very* carefully, and often in Assembler Language!

[127] Another memory-saving technique was *overlay*, which we'll describe briefly in Section 38.9.

[128] Partitioning uses address translation to allow more than one operating system to run in a single physical memory, each behaving as if its set of “real” addresses starts at zero.

- in 64-bit mode, all 64 bits form the Effective Address.



Remember:

An Effective Address is not the same as the contents of a register, even though it may be derived from the contents of one or more registers.

The areas of addressability for the three addressing modes are sketched in Figure 149.

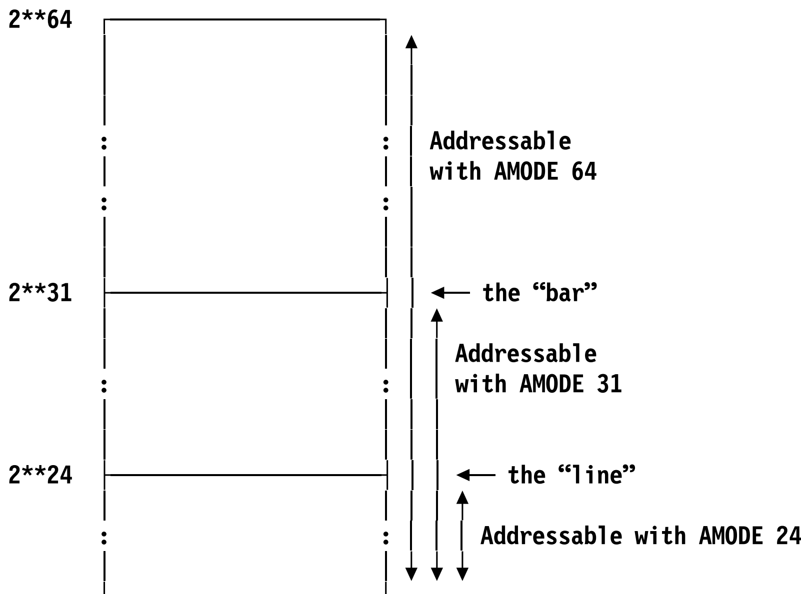


Figure 149. Areas of memory addressed by three AMODEs

Instructions that place or update addresses in the general registers are called "modal" instructions, because the result depends on the addressing mode. We'll see some examples in Section 20.3.

Effective Address addressing-mode considerations

- In 24-bit or 31-bit addressing modes, 40 or 33 high-order bits of the 64-bit Effective Address (respectively) are set to zero.
- If a value in a general register is used for addressing, its high-order bits are not set to zero (as for generated addresses), but are ignored.

The CPU's current addressing mode is determined by two bits in the Program Status Word, "Basic addressing mode" and "Extended addressing mode", illustrated in Figure 150.

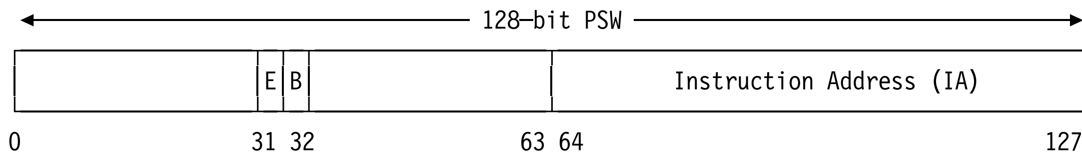


Figure 150. System z PSW showing addressing-mode bits

The meanings of the E and B bit settings are shown in Table 99 on page 309.

E	B	Addressing mode
0	0	24-bit mode
0	1	31-bit mode
1	0	Invalid combination
1	1	64-bit mode

Table 99. PSW addressing-mode bits

Almost all instructions that reference operands in memory depend in some way on the current addressing mode; and instructions that update addresses in registers also depend on the addressing mode. These are called *modal* instructions. Other instructions (like AR) are called *non-modal* because their results are independent of addressing modes.

In Section 38 we will see instructions used to change addressing modes, and show why attention to addressing modes can be very important — and very useful..

Exercises

20.2.1.(1)+ Suppose c(GG1)=X'00000000 82006A04' and c(GG2)=X'00000000 FFFF8200'. An RXY-type instruction at address X'629D58' looks like this:

opcode	A	2	1	X'A06'	X'04'	opcode

What Effective Address does it generate in 24-bit addressing mode? In 31-bit addressing mode? In 64-bit addressing mode?

20.2.2.(2) Repeat Exercise 20.1.3, showing how the generated Effective Addresses depend on the addressing mode.

20.3. Load Address Instructions

The name “Load Address” is misleading: the instruction loads a register (but not from memory or another register), and its operand may or may not be an address: the Effective Address of the second operand is loaded into the R_1 register. Thus, it might more properly be named “Load Effective Address”.

Although we normally wouldn't consider it a logical instruction, Load Address is often classified that way. The three instructions are listed in Table 100.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
41	LA	RX	Load Address	E371	LAY	RXY	Load Address
C00	LARL	RIL	Load Address Relative Long				

Table 100. Load Address instructions

LA and LAY are RX- and RXY-type instructions, and LARL generates the Effective Address from its address and the 32-bit RI_2 operand, as described in Section 20.1.3 on page 305. In each case, the Effective Address replaces the contents of GR R_1.

The affected parts of GR R_1 depend on the CPU's current *addressing mode*. As noted in Section 20.2, some of the high-order bits of the Effective Address may be set to zero.

Suppose the following LAY instruction is at address X'003B6D0E', and addressability has been established. Then if we execute

```
         LAY   0,-1                 Put -1 in register 0 (?)
```

the result depends on the addressing mode:

- in 24-bit mode, the Effective Address is X'00FFFFFF', and the high-order 32 bits of GG0 are unchanged.
- in 31-bit mode, the Effective Address is X'7FFFFFFF', and the high-order 32 bits of GG0 are unchanged.
- in 64-bit mode, the Effective Address is X'FFFFFFFFFFFFFFFF', and the high-order 32 bits of GG0 are changed.

Modal Instructions

LA, LAY, and LARL are *modal* instructions: the resulting Effective Address depends on the addressing mode.

In any addressing mode, a nonnegative integer “n” between 0 and 4095 can be placed in a register by executing

```
         LA    r,n(0,0)
```

where the displacement contains the constant “n”. Instead of writing

```
         L     2,=F'1'
```

requiring 8 bytes (4 for the instruction and 4 for the constant generated by the literal), or

```
         LH    2,=H'1'
```

requiring 6 bytes, we can write either

```
         LA    2,1          or        LA    2,1(0,0)
```

This requires only 4 bytes *and* less execution time, because no memory access is required.

Large *signed* integer values can be placed in a 64-bit register using LAY if the addressing mode is 64-bit, as shown in Figure 151.

```
         LAY   0,500000          c(GR0) = +500000
         LAY   1,-500000         c(GG1) = -500000 (64-bit mode only!)
```

Figure 151. Loading integer constants with the LAY instruction

You can also use values assigned to absolute symbols:

```
HalfMiln Equ   500000            Value = +500000
         LAY   2,HalfMiln        c(GR2) = +500000
         LAY   2,-HalfMiln       c(GG2) = -500000 (64-bit mode only!)
```

This can often eliminate the need for constants in memory and the storage references needed to access them.

For signed arithmetic values, it can be safer to initialize a register with one of the arithmetic immediate instructions described in Section 21.2 on page 321.

Be Very Careful!

The Effective Address will depend on the addressing mode! LAY 0,-1 generates X'00FFFFFF' in 24-bit addressing mode, and X'7FFFFFFF' in 31-bit mode.

For example, see Exercise 20.3.8.

Because LA and LAY do not affect the CC, we can clear a register without disturbing a CC setting that may be required at a later point in the program. For example, suppose we wish to add c(A) and c(B) and clear the result to zero if it overflows, without changing the CC set by the addition. These two instruction sequences will work:

```
         L     0,A                          L     0,A
         A     0,B                          A     0,B
         BNO   ST                           BNO   ST
         LA    0,0                          L     0,=F'0'
ST       ST    0,Answer            ST       ST    0,Answer
```

Because the LA instruction computes an Effective Address, it also provides a simple way to increment a number in a register (other than register 0) by a small positive amount. We put the increment into the displacement, and use the same register for the R_1 and B_2 digits. For example,

```
         LA    4,17(0,4)
```

increases the contents of GR4 by 17, *if* the original value in GR4 is not corrupted. For example, in 24-bit addressing mode, c(GR4) must lie between -17 and $2^{24}-18$. Using LA to increment register contents is usually limited to cases where the quantity being incremented is an address or a reasonably small integer.

Be Careful!

Don't use a Load Address instruction to increment a negative number, or a number large enough that the result might be affected by the current addressing mode.

Suppose we want to perform the shifting operation described in Figures 113 and 114 on page 248, where we wanted to shift the word at **N** to the right enough places so that its rightmost bit is a 1-bit. Now, however, we also require that the number of positions shifted be stored at the halfword named **Count**.

```
         L     4,N                Get integer to shift
         LA    3,1                Set GR3 to 1
         LCR   3,3                Initial shift count set to -1
Shift    SRDL  4,1                Shift a bit into GR5
         LTR   5,5                Test sign of GR5
         LA    3,1(0,3)           Increment GR3 by 1
         BNM   Shift              Branch if GR5 not negative
         SLDL  4,1                Move bit back into place
         STH   3,Count            Store shift count
```

Figure 152. Counting number of shifts to make rightmost bit a 1-bit

By setting the shift count to -1 initially, we guarantee that the correct value will be in GR3 when we exit from the loop. The first time the LA instruction is executed, the result in GR3 will be zero. The placement of the LA instruction between the LTR and the ensuing BNM shows that no change is made to the CC; normally, we would place the LTR just before the BNM because the relation between the two is then clearer to the program's readers.

A third use of the LA instruction, and possibly the most important, is to generate addresses for operands in memory. For example, we may require the address of some operand to be in a given register while executing a segment of code. Suppose we want to add three integers, and branch after all additions are completed to **NoErr** if no overflow occurs, and to **Err1** if one or more overflows occur. Let the integers to be added be stored in successive words beginning at **QQ**.

```
         LA    9,NoErr            Set branch address for no errors
         L     2,QQ               Get first integer
         A     2,QQ+4             Add second integer
         BNO   OK1                Branch if no overflow
         LA    9,Err1             Set branch address for overflow
OK1      A     2,QQ+8             Add third integer
         BNOR  9                  Branch if no or one overflow
         B     Err1               Branch, some addition overflowed
```

Figure 153. Using LA to set a branch address

The last unconditional branch instruction could also be written

```
         BO    Err1
```

without affecting the operation of the code, since that instruction is reached only if the branch condition for the immediately preceding instruction is not met. By specifying an unconditional branch it is clear that the branch must always be taken if it is reached.

There is an important assumption in Figure 153 on page 311 regarding the two LA instructions: the locations named **NoErr** and **Err1** must be addressable, since the LA instruction simply performs the address computation specified by the base and displacement assigned by the Assembler. It's sometimes easy to forget that symbols used in LA instructions must be addressable, since no direct reference is being made to a memory location: only an Effective Address is being generated, and no checks are made for the validity of that address.

The addressability limitations of LA can often be overcome using LAY or LARL.

Exercises

20.3.1.(1) The LARL instruction has a signed 32-bit RI_2 immediate operand. Why can LARL not be used to load the R_1 register with a large even integer value?

20.3.2.(1) If the CPU is executing in 24-bit addressing mode, show how the LA instruction can be used as a masking instruction, producing the same result in a register as

```
         N     reg,=A(X'FFFFFF')
```

20.3.3.(1)+ Can the first machine instruction in Figure 153 on page 311 be written

```
         LA    9,A(NoErr) ?
```

20.3.4.(1)+ Can the first machine instruction in Figure 153 on page 311 be written

```
         LA    9,=A(NoErr) ?
```

20.3.5.(2)+ The following two instructions usually have an equivalent effect:

```
         LA    9,NoErr          c(GR9) = A(NoErr)
         L     9,=A(NoErr)      c(GR9) = A(NoErr)
```

Under what circumstances would you use one in preference to the other? Under what circumstances would the two *not* be equivalent?

20.3.6.(2)+ Suppose there is a number between 0 and 7 in GR5, and you want to place into GR8 a single bit whose position within the low-order byte of that register is given by the number in GR5. Thus, if GR5 contains X'00000006', GR8 should contain X'00000002'. A student claimed that the following code sequence does the job; prove or disprove that claim.

```
         LA    8,X'100'(0,5)
         SRL   8,1(8)
```

20.3.7.(2) Discuss the differences between

```
         LA    x,number(0,x)
and      AH    x,=H'number'
```

as techniques for incrementing the contents of register GRx by a small positive integer "number". Under what circumstances would the result be different, and in what ways? Will it work for all values of GRx? Which values will work, and which values won't? What differences may be required if "number" is defined by an EQU statement like this?

```
number   EQU   29
```

20.3.8.(3) In Figure 151 on page 310, what will the assembled instructions look like? How will the results depend on the current addressing mode?

20.3.9.(2) Suppose you execute these two instructions in 24-bit addressing mode:

```
         L     6,=A(X'FFFFFF')
         LA    6,2(,6)
```

What value will be in GR6?

What value will be in GR6 if the first instruction had been written

```
         L     6,=A(X'FFFFFF00')      ?
```

20.3.10.(2) In Figure 152 on page 311, we might want to initialize GR3 to −1 using

```
         LAY   3,-1
```

What reasons might be given for *not* using LAY?

20.3.11.(2)+ If x and y are numbers between 1 and 15, what are the differences between these two instructions?

```
         LR    x,y          and          LA    x,0(0,y)
```

20.3.12.(3)+ Suppose GR15 contains one of the values 0, 4, 8, or 12. Depending on c(GR15), you want to branch to **A**, **B**, **C**, or **D** respectively.[129] For a program with a base register providing addressability to the code, you might write

```
         B     BList(15)        Branch into table of branches
BList    B     A                Branch if c(GR15) = 0
         B     B                Branch if c(GR15) = 4
         B     C                Branch if c(GR15) = 8
         B     D                Branch if c(GR15) = 12
```

Suppose your program has *no* base register to provide addressability for the code. How can you accomplish this task?

20.3.13.(1) Why can't use use LA or LAY to increment a small nonnegative number in GR0?

20.3.14.(2)+ We are given these two definitions of the symbol `Number`:

1. `Number  DC    X'1234'`
2. `Number  DC    X'ABCD'`

Assuming 24-bit addressing mode: what hexadecimal value is left in GR10 by this instruction sequence?

```
         LH    10,Number
         SLL   10,8
         LA    10,0(10,0)
         SRL   10,8
```

Now, in 31-bit addressing mode, what hexadecimal value is left in GR10 for each definition of `Number`?

20.3.15.(2)+ Suppose the contents of general registers 0, 1, and 2 are given by c(GR0)=X'2112E6D8', c(GR1)=X'9B017822', and c(GR2)=X'00FFFF00'. What Effective Address is generated in 24-bit addressing mode for each of the following addressing halfwords?

1. X'00FE'
2. X'1AF9'
3. X'2109'

Now, find the Effective Addresses generated in 31-bit addressing mode.

20.3.16.(2)+ Suppose this instruction is at address X'543B6D0E':

[129] This is a common convention for handling a "return code" from a called subroutine.

```
         LAY   0,*
```

What Effective Address will be generated in (1) 24-bit, (2) 31-bit, and (3) 64-bit addressing mode?

20.3.17.(3)+ A programmer claims that you can test whether adding a length in GR1 to an existing address in GR2 will cross a known power-of-two boundary with the instructions shown below. Write a test program with various input values to test his assertion.

```
         LAY   1,-1(1,2)
         XR    1,2
         N     1,Mask
         JNZ   Crossed          Branch if adding crosses
         - - -
Mask     DC    A(-8192)         Negative of power of 2 boundary
```

20.4. 64-Bit Virtual Addresses

We saw in Figure 22 on page 68 how 31-bit virtual addresses are divided into shorter components for mapping with translation tables into real addresses. The same technique is used for 64-bit virtual addresses, except that an additional 33 high-order bits must be mapped. This is illustrated in Figure 154.

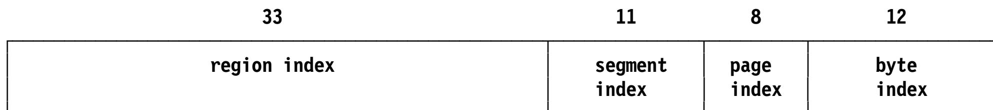


Figure 154. 64-bit Virtual Address

Because a 33-bit translation table would be extremely large, the region index is subdivided into three portions, called “region first“, “region second”, and “region third” indexes, for which the mapping tables are more manageable. This is sketched in Figure 155.

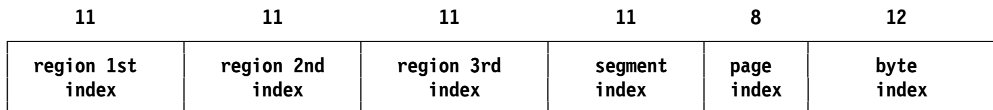


Figure 155. 64-bit Virtual Address with Region Indexes

Fortunately, these details are handled by the operating system so we can focus on our applications.

20.5. Summary

In this section, we discussed addressing modes and the three instructions shown in Table 101.

Function	Instruction	Result in R_1 general register		
		AMode = 24	AMode = 31	AMode = 64
Load Address (based) Load Address (relative)	LA LAY LARL	Effective Address in bits 40-63; zero in bits 32-39; bits 0-31 unchanged.	Effective Address in bits 33-63; zero in bit 32; bits 0-31 unchanged.	Effective Address in bits 0-63.

Table 101. Load Address instructions described in this section

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
LA	41

Mnemonic	Opcode
LARL	C00

Mnemonic	Opcode
LAY	E371

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
41	LA

Opcode	Mnemonic
C00	LARL

Opcode	Mnemonic
E371	LAY

Terms and Definitions

addressing mode
One of three modes supported by System z that determines the length of an Effective Address.

AMode
An abbreviation for "addressing mode".

DH
In an instruction supporting 20-bit displacements, the 5th byte of the instruction containing the signed **H**igh-order 8 bits of the displacement.

DL
In an instruction supporting 20-bit displacements, the unsigned **L**ow-order 12 bits of the displacement.

modal instruction
An instruction that places or updates addresses in the general registers, with results that depend on the *addressing mode*.

relative address
An Effective Address determined by an offset *relative* to an instruction containing an RI_2 operand.

21. Immediate Operands

```
 2222222222      11
222222222222    111
22        22   1111
          22     11
          22     11
         22      11
       22        11
     22          11
   22            11
 22              11
222222222222  1111111111
222222222222  1111111111
```

In Section 4.2 on page 51 we saw the five basic instruction classes introduced with System/360. The only class with immediate data was the SI-type instructions, where a byte of data in the instruction operated on, or was stored into, a byte in memory, as sketched in Figure 14 on page 52. We'll learn more about those in Section 23.

Many new instructions include “immediate’ data that is part of the instruction, rather than being in memory. Thus, it is “immediately” available. Most immediate operands work with data in the general registers, rather than memory: Figure 156 shows how immediate operands in RI- and RIL-type instructions interact with data in registers.[130] You may want to compare it to Figure 14 on page 52.

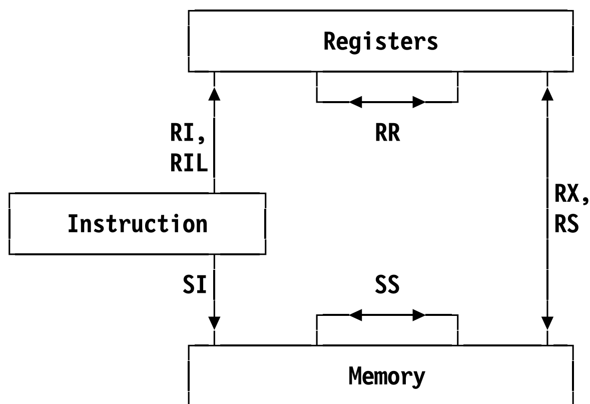


Figure 156. Instruction classes, including RI, RIL

In early processors, the relative speeds of memory accesses and instruction execution using memory operands were nearly the same. As processor speeds have increased, instructions can often be completed in much less time than it takes to access memory operands. As this speed difference has grown, the relative cost of memory accesses has also grown, despite many methods

[130] Because z/Architecture continues to evolve, you should check the *z/Architecture Principles of Operation* regularly; some newer instructions operate on immediate data in the instruction *and* data in memory.

providing intermediate stages of "buffering", using special internal cache memories. Caches can help reduce, but not eliminate, the speed difference.

Because memory accesses in many applications refer to constant data, instructions containing these constants provide immediate access to the data without additional memory references. The resulting improvements in application performance have shown the value of these relative-immediate instructions.

Two immediate-operand lengths are supported. In RI-type instructions, the I_2 immediate operand occupies a halfword, the last 16 bits of the 32-bit instruction.

opcode	R_1	Op	I_2

Table 102. RI-type instruction

In RIL-type instructions, the immediate operand I_2 occupies a word, the last 32 bits of the 48-bit instruction.

opcode	R_1	Op	I_2

Table 103. RIL-type instruction

Several of the instructions we'll examine use the last two letters of the instruction mnemonic to indicate a specific portion of the R_1 register, with combinations of "H" and "L". The first letter refers to the **H**igh half of a 64-bit register or the **L**ow half of the register. Similarly, the second letter refers to the **H**igh halfword or the **L**ow halfword of the half of the register specified by the first letter.[131] This is illustrated in Figure 157.

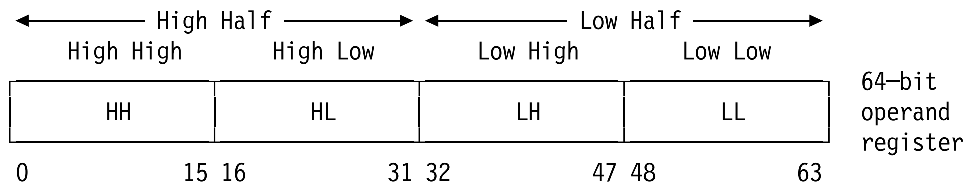


Figure 157. Four halfwords in a 64-bit general register

Another way of describing this:

HH High Half's High Half (bits 0-15)
HL High Half's Low Half (bits 16-31)
LH Low Half's High Half (bits 32-47)
LL Low Half's Low Half (bits 48-63)

Other instructions with 32-bit immediate operands end in the letters "H" or "L" meaning the **H** or **L**ow half of the register, followed by "F" to indicate that the immediate operand is a fullword.

We'll now investigate these instructions in three groupings: insert and load, arithmetic, and logical.

[131] See the comments at the start of Section 14, on page 178.

21.1. Insert and Load Instructions with Immediate Operands

21.1.1. Logical-Immediate Insert Instructions

The insert group of logical-immediate instructions is summarized in Table 104.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
C08	IIHF	RIL	Insert Logical Immediate (high) (64←32)	C09	IILF	RIL	Insert Logical Immediate (low) (64←32)
A50	IIHH	RI	Insert Logical Immediate (high high) (64←16)	A51	IIHL	RI	Insert Logical Immediate (high low) (64←16)
A52	IILH	RI	Insert Logical Immediate (low high) (64←16)	A53	IILL	RI	Insert Logical Immediate (low low) (64←16)

Table 104. Insert-Immediate instructions

The sketch in Figure 158 shows the operation of these six instructions. For example, IIHF inserts its 32-bit (**F**ullword) immediate operand into the high half of GG R_1.

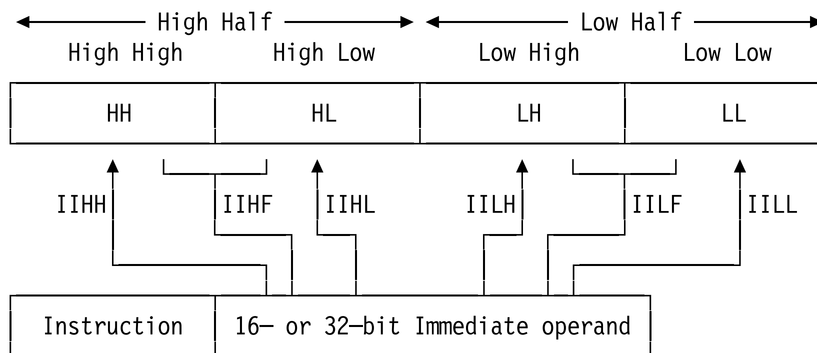


Figure 158. Operation of six Insert Immediate instructions

The insert-immediate operations are similar to the capabilities of the ICM and ICMH instructions that refer to storage operands. For example, these two instructions have the same result:

```
         ICM   5,B'1100',=C'LH'   Insert 'LH' into bits 0-15 of GR5
         IILH  5,C'LH'            The same with an immediate operand
```

except that IILH avoids a memory reference. Similarly, these two are equivalent:

```
         ICMH  3,B'1111',=F'-3'   Insert -3 into bits 0-31 of GG3
         IIHF  3,-3               The same with an immediate operand
```

You can think of the IILF instruction as though it's a “Load Immediate” instruction:[132]

```
         IILF  11,123456789       has the same result as...
         L     11,=F'123456789'   so you could even think of it as L
***      LI    11,123456789       ...but not as LI!
```

These instructions let you insert 16- or 32-bit operands into any halfword or word portion of a general register without disturbing other parts of the register.

21.1.2. Arithmetic- and Logical-Immediate Load Instructions

These instructions are listed in Table 105 on page 319.

[132] But you could implement your own LI macro instruction using the macro instruction capabilities of the Assembler.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
A78	LHI	RI	Load Halfword Immediate (32←16)	A79	LGHI	RI	Load Halfword Immediate (64←16)
C01	LGFI	RIL	Load Immediate (64←32)				
C0E	LLIHF	RIL	Load Logical Immediate (high) (64←32)	C0F	LLILF	RIL	Load Logical Immediate (low) (64←32)
A5C	LLIHH	RI	Load Logical Immediate (high high) (64←16)	A5D	LLIHL	RI	Load Logical Immediate (high low) (64←16)
A5E	LLILH	RI	Load Logical Immediate (low high) (64←16)	A5F	LLILL	RI	Load Logical Immediate (low low) (64←16)

Table 105. Load and insert instructions with immediate operands

The LHI, LGHI, and LGFI instructions are *arithmetic* load operations, where the I_2 immediate operand is sign-extended from 16 to 32 bits or from 32 to 64 bits, as required by the R_1 register length. They operate just like the corresponding LH, LGH, and LGF instructions, except that the second operand is found in the I_2 field of the instruction rather than in memory. For example, compare the operation of the LHI instruction in Figure 159 with the operation of LH in Figure 62 on page 183:

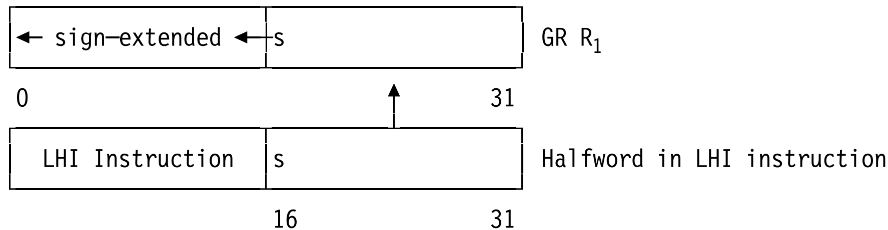


Figure 159. Operation of LHI instruction

A valuable application of instructions like LHI involves symbolically-defined constants. Suppose you have a table of data items, and you define a symbol **NItems** whose value is the number of items:

```
Table    DS    0F                    Start of table
Item1    DS    CL(ItemLen)           Each item has length 'ItemLen'
               - - -                 Space for more similar items
TableEnd DS    0X                    End of the table
*
NItems   Equ   ((TableEnd-Table)/ItemLen)   Number of items in table
```

Then, you can place the count of data items into GR8 using LHI:

```
         LHI   8,NItems              Initialize item counter
```

Defining symbols like **NItems** symbolically means that if the table expands or contracts, you need only reassemble the program and the value of **NItems** will be recalculated automatically.

LGHI extends its 16-bit operand to 64 bits in GG R_1, as shown in Figure 160.

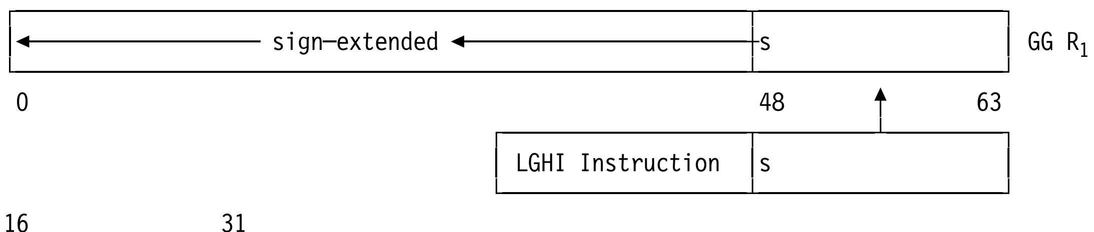


Figure 160. Operation of LGHI instruction

Similarly, the LGFI instruction extends its 32-bit operand to 64 bits, as shown for LGF in Figure 73 on page 195.

Suppose c(GG9)=X'12345678 00000000'; then

```
         LHI   9,X'CBA9'          Load a halfword-immediate operand
```

will set the rightmost 32 bits of GR9 to X'FFFFCBA9', leaving the high-order 32 bits of GG9 unchanged. Now suppose c(GG9)=X'12345678 9ABCDEF0'; the other two load-immediate instructions give the sign-extended results shown in Figure 161:

```
         LGHI  9,X'CBA9'         c(GG9)=X'FFFFFFFF FFFFCBA9' extend 1-bit
         LGFI  9,X'789ABCDE'     c(GG9)=X'00000000 789ABCDE' extend 0-bit
```

Figure 161. Examples of load-immediate instructions

The other six logical instructions have an unusual property: the I_2 immediate operand is placed in the proper 16 or 32 bits of the 64-bit general register, and (unlike the insert-immediate instructions) the rest of the *entire* register is set to zero! The Load Logical instructions we discussed in Section 14.11 on page 195 did zero-extension only on the left, rather than also zeroing bits to the right of the loaded operand.

The following figure pictures the operation of these six logical load instructions.

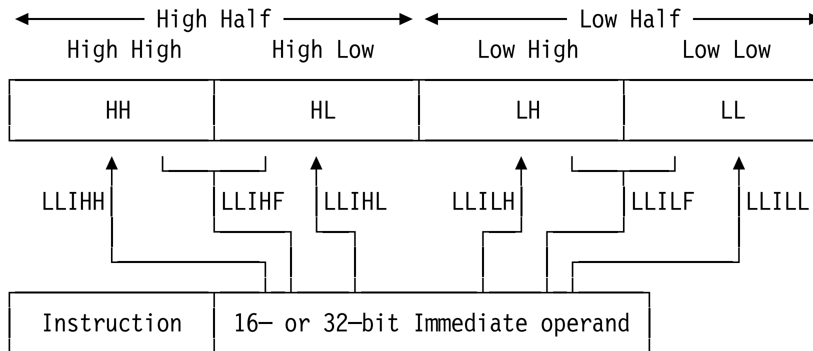


Figure 162. Operation of six logical load instructions

In each case, after the I_2 operand has been loaded into the specified part of GG R_1, the rest of the register is set to zero. For example, if c(GG9)=X'FEDCBA9876543210', executing each of the following instructions will change GG9 as indicated:

```
         LLIHF 9,X'13579BDF'     c(GG9)=X'13579BDF 00000000'
         LLILF 9,X'FDB97531'     c(GG9)=X'00000000 FDB97531'
         LLIHH 9,X'2468'         c(GG9)=X'24680000 00000000'
         LLIHL 9,X'2468'         c(GG9)=X'00002468 00000000'
         LLILH 9,X'2468'         c(GG9)=X'00000000 24680000'
         LLILL 9,X'2468'         c(GG9)=X'00000000 00002468'
```

The Load Logical Immediate instructions are useful whenever you need to place a value into part of a general register and set the rest of the register to zero, and they help you avoid unnecessary clearing of the target register. For example, if the LLIHF instruction was not available and you wanted to load X'13579BDF' into the high-order 32 bits of GG9 (as in the first instruction above), you would have to do something like

```
         L     9,=F'13579BDF'    c(GR9)=X'13579BDF'
         SLLG  9,9,32            c(GG9)=X'13579BDF 00000000'
```

requiring both a memory reference and an extra instruction. Similarly, to get the result of the LLILH instruction above, you would have to do these two instructions

```
         SGR   9,9               Set GG9 to zero
         IILH  9,X'2468'         c(GG9)=X'00000000 24680000'
```

which uses one of the immediate-operand instructions. Or, another use could have been

```
         SGR   9,9                Set GG9 to zero
         ICM   9,B'1100',=X'2468' c(GG9)=X'00000000 24680000'
```

again requiring an extra instruction and a memory access.

Exercises

21.1.1.(1) What will the Assembler do if you write `LHI 0,76543` ? What will be placed in GR0?

21.1.2.(1) What is the difference between an I_2 operand and an RI_2 operand?

21.2. Arithmetic Instructions with Immediate Operands

The arithmetic instructions with immediate operands can be arranged in three groups:

- add and subtract instructions
- compare instructions
- multiply instructions

These instructions can be arranged into very regular patterns, like the related RX-type instructions we saw in Section 16.

21.2.1. Arithmetic-Immediate Add and Subtract Instructions

The four arithmetic and four logical instructions in this group are shown in Table 106.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
A7A	AHI	RI	Add Halfword Immediate (32←16)	A7B	AGHI	RIL	Add Halfword Immediate (64←16)
C29	AFI	RIL	Add Immediate (32)	C28	AGFI	RIL	Add Immediate (64←32)
C2B	ALFI	RIL	Add Logical Immediate (32)	C2A	ALGFI	RIL	Add Logical Immediate (64←32)
C25	SLFI	RIL	Subtract Logical Immediate (32)	C24	SLGFI	RIL	Subtract Logical Immediate (64←32)

Table 106. Arithmetic-immediate add and subtract instructions

These instructions are very useful: they can replace most memory references to constants and literals. Consider the example in Figure 88 on page 223 where we add the first N odd numbers, but now we use immediate values instead of literals. Three storage references have been replaced by immediate operands in the statements marked with * in the comment field.

```
         LHI   4,1         * c(GR4) = accumulated sum
         LR    7,4           c(GR7) = count of additions
Test     CH    7,NN          Compare count to c(NN)
         BE    Store         Branch if equal, N terms added
         LR    0,7           Compute next odd integer
         AR    0,0           Counter + counter = 2N
         AHI   0,1         * Add 1, giving next odd term
         AR    4,0           Add term to sum
         AHI   7,1         * Increment count by 1
         B     Test          Branch back to see if finished
Store    ST    4,SUM         Store result
```

Almost every previous example using a halfword or word literal can be replaced by an immediate operand. This saves both execution time and the bytes needed for the storage operand.

21.2.2. Arithmetic-Immediate Compare Instructions

The four arithmetic and two logical instructions in this group are shown in Table 107.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
A7E	CHI	RI	Compare Halfword Immediate (32←16)	A7F	CGHI	RI	Compare Halfword Immediate (64←16)
C2D	CFI	RIL	Compare Immediate (32)	C2C	CGFI	RIL	Compare Immediate (64←32)
C2F	CLFI	RIL	Compare Logical Immediate (32)	C2E	CLGFI	RIL	Compare Logical Immediate (64←32)

Table 107. Arithmetic-immediate compare instructions

Suppose you must examine a character in storage to see if it is a special character, or a letter or digit, and retain the character in GR0 for further processing. (Remember that letters and digits in the EBCDIC representation have values greater than X'80'.) You could write the test like this:

```
         LLC   0,Char              Get character, clear rest of GR0
         CHI   0,X'80'             Test for special character
         BNH   Special             Special if representation <= X'80'
```

The LLC instruction was illustrated in Figure 75 on page 197.

It helps to remember that these compare-immediate instructions always refer to operands in registers, never in memory:

```
         CH    2,NN                Compare to halfword in memory...
*        CHI   2,NN                ... but this would fail if assembled
         - - -
NN       DC    H'42'
```

21.2.3. Arithmetic-Immediate Multiply Instructions

Table 108 lists the two arithmetic multiply-immediate instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
A7C	MHI	RI	Multiply Halfword Immediate (32←16)	A7D	MGHI	RI	Multiply Halfword Immediate (64←16)

Table 108. Arithmetic-immediate multiply instructions

There are no multiply-immediate instructions with 32-bit operands.[133] This is rarely a problem, because you can use instructions like IILF or LGFI to put a 32-bit operand into a temporary register. For example, if the product and operands are small enough you can use MHI:

```
         L     1,Operand1          Get a number to be multiplied
         MHI   1,36                Multiply by 36, product in GR1
```

and if the product and operands are larger, you can use IILF:

```
         L     1,Operand2          Get another number to be multiplied
         IILF  15,629036721        Put multiplier temporarily in GR15
         MR    0,15                Form long product in (GR0,GR1)
```

[133] At the time of this writing. But new instructions are added regularly to the System z architecture, so check the *Principles of Operation*.

Exercises

21.2.1.(1) Why is there a SLFI instruction, but no SFI instruction?

21.2.2.(2) Do Exercise 18.2.7 on page 269, using immediate-operand instructions and no literals.

21.3. Logical Operations with Immediate Operands

As we have seen, the last two letters of these instruction mnemonics refers to a word or halfword in part of a 64-bit general register. The three logical operations are AND, OR, and XOR.

The portions of the first operand in GG R_1 not involved in the operation of these instructions is not affected, and remain unchanged. This was shown in Figure 158 on page 318, where the "Insert Immediate" instructions involve either 16 or 32 bits of the register, and the remaining bits are unchanged. (The "Load Immediate" instructions *do* clear the remaining fields of the register!)

21.3.1. Logical-Immediate AND Instructions

The AND group of logical-immediate instructions is summarized in Table 109.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
C0A	NIHF	RIL	AND Immediate (high) (64←32)	C0B	NILF	RIL	AND Immediate (low) (64←32)
A54	NIHH	RI	AND Immediate (high high) (64←16)	A55	NIHL	RI	AND Immediate (high low) (64←16)
A56	NILH	RI	AND Immediate (low high) (64←16)	A57	NILL	RI	AND Immediate (low low) (64←16)

Table 109. AND-immediate instructions

The last example in Section 19.3 uses a bit mask in memory. We can improve it by using an immediate operand in an NILL instruction.

```
L     1,DataWord        B'aaaaaaaaabbbbcccccccccccccdddddd'
SRL   1,6               B'000000aaaaaaaaabbbbccccccccccccc'
NILL  1,X'1FFF'         B'0000000000000000000ccccccccccccc'
ST    1,Third           Store Result
```

Figure 163. Extracting an unsigned integer value using AND Immediate

The last example in Section 19.4 uses a bit mask in memory. We can also improve it using an NILF immediate operand:

```
L     0,DataWord        Get 4 packed integers
NILF  0,X'FFF8003F'     Clear a space for third (c's)
L     1,NewThird        Get new value of third integer
SLL   1,6               Shift into proper position
OR    0,1               'OR' into place in GR0
ST    0,DataWord        Store new data word
```

Figure 164. Inserting a new integer value using AND Immediate

21.3.2. Logical-Immediate OR Instructions

The OR group of logical-immediate instructions is summarized in Table 110 on page 324.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
C0C	OIHF	RIL	OR Immediate (high) (64←32)	C0D	OILF	RIL	OR Immediate (low) (64←32)
A58	OIHH	RI	OR Immediate (high high) (64←16)	A59	OIHL	RI	OR Immediate (high low) (64←16)
A5A	OILH	RI	OR Immediate (low high) (64←16)	A5B	OILL	RI	OR Immediate (low low) (64←16)

Table 110. OR-immediate instructions

Suppose you want to set the sign bit of GG8 to a 1-bit. You can use either of these:

```
         OIHH  8,X'8000'         Set sign bit to 1
         OIHF  8,X'80000000'     Set sign bit to 1
```

but OIHF is 6 bytes long while OIHH is only 4 bytes long.

21.3.3. Logical-Immediate XOR Instructions

The XOR group of logical-immediate instructions is summarized in Table 111.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
C06	XIHF	RIL	XOR Immediate (high) (64←32)	C07	XILF	RIL	XOR Immediate (low) (64←32)

Table 111. XOR-immediate instructions

You might wonder why there are no XIHH, XIHL, XILH, and XILL instructions, like those for the 16-bit operands of the logical-immediate AND and OR instructions. (See Exercise 21.3.1.)

The example in Figure 141 on page 292 uses AND, OR, and XOR instructions referring to operands in memory. We can rewrite it to use immediate operands:

```
         L     0,DataWord          Get integers
         OILF  0,X'0007FFC0'     * Set third-integer space to all 1's
         XILF  0,X'0007FFC0'     * Now set them to zeros
         L     1,NewThird          Load new value for third integer
         SLL   1,6                 Move to correct position
         NILF  1,X'0007FFC0'     * Make sure there are no extra bits
         OR    0,1                 Insert the new third value
         ST    0,DataWord          Store updated result
```

Figure 165. Data masking using immediate operands

We can improve this example to reduce the possibility of typographic errors, by defining the mask symbolically:

```
Int3Mask Equ   X'0007FFC0'         Mask for isolating the 3rd integer
         L     0,DataWord          Get integers
         OILF  0,Int3Mask          Set third-integer space to all 1's
         XILF  0,Int3Mask          Now set them to zeros
         L     1,NewThird          Load new value for third integer
         SLL   1,6                 Move to correct position
         NILF  1,Int3Mask          Make sure there are no extra bits
         OR    0,1                 Insert the new third value
         ST    0,DataWord          Store updated result
```

Figure 166. Data masking using a symbolically defined immediate operand

This technique is recommended whenever one value must be used in several instructions. If you mistype the mask value, it needs correcting in only one place.

Exercises

21.3.1.(2)+ Explain why there is actually no need for the four XOR halfword-immediate instructions XIHH, XIHL, XILH, and XILL.

21.3.2.(2)+ Show why the "Halfword" forms of the AND-immediate logical NIxx instructions (like NILH, etc.) are unnecessary.

21.3.3.(2)+ Show why the "Halfword" forms of the OR-immediate logical OIxx instructions (like OIHL, etc.) are unnecessary.

21.3.4.(1) Use instructions with immediate operands to set the high-order byte of GR1 to zero.

21.3.5.(1) Use instructions with immediate operands to invert the sign bit of GG7.

21.3.6.(1) Use instructions with immediate operands to round c(GR2) to the next higher multiple of 16, if it is not already a multiple of 16.

21.3.7.(2) A programmer wanted to test the value of some bits in GR3, and wrote these instructions:

```
         NILL  3,X'00F0'     Isolate the 4 interesting bits
         BZ    AllZeros      Branch if all 4 bits were zero
         CH    3,=X'0070'    Check if leftmost bit is 1
         BNL   BitWas1       Branch if that bit was 1
         - - -               Test other values
```

What value will be in GR3 when control arrives at the instruction named **BitWas1**?

21.3.8.(2)+ A programmer wanted to extract the six low-order bits of GR4, and considered these three sequences of instructions:

```
(1)      N     4,=X'0000003F'

(2)      SLL   4,26
         SRL   4,26

(3)      SRDL  4,26
         SR    4,4
         SLDL  4,26
```

Criicize each sequence in terms of its simplicity and/or efficiency, and suggest a single instruction to use in place of each.

21.3.9.(2)+ A friend of the programmer in Exercise 21.3.8 suggested using an instruction with an immediate operand:

```
         NILL  4,X'003F'
```

Is his solution acceptable? Explain why or why not.

21.4. Summary

The immediate-operand instructions described in this section can provide savings in three ways:

1. they eliminate the need to access operands from storage,
2. they save the space that those operands needed, and
3. they help eliminate the need for base registers that might have been required to address those operands.

The load- and insert-immediate instructions are summarized in Table 112 on page 326. The insert-immediate instructions don't affect any part of the R_1 register other than the bit positions where the immediate operand has been inserted.

Operation	Operand 1	32 bits		64 bits	
	Operand 2	16 bits	32 bits	16 bits	32 bits
Arithmetic Load		LHI		LGHI	LGFI
Logical Load				LLIHH LLIHL LLILH LLILL	LLIHF LLILF
Insert				IIHH IIHL IILH IILL	IIHF IILF

Table 112. Load and insert instructions with immediate operands

The arithmetic-immediate instructions are summarized in Table 113.

Operation	Operand 1	32 bits		64 bits	
	Operand 2	16 bits	32 bits	16 bits	32 bits
Arithmetic Add/Subtract		AHI	AFI	AGHI	AGFI
Logical Add/Subtract			ALFI SLFI		ALGFI SLGFI
Arithmetic Compare		CHI	CFI	CGHI	CGFI
Logical Compare			CLFI		CLGFI
Multiply		MHI		MGHI	

Table 113. Arithmetic instructions with immediate operands

The logical-immediate instructions are summarized in Table 114. The logical-immediate instructions don't affect any part of the R_1 register other than the bit positions where the immediate operand has been ANDed, ORed, or XORed.

Operation	Operand 1	32 bits		64 bits	
	Operand 2	16 bits	32 bits	16 bits	32 bits
AND				NIHH NIHL NILH NILL	NIHF NILF
OR				OIHH OIHL OILH OILL	OIHF OILF
XOR					XIHF XILF

Table 114. Logical instructions with immediate operands

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
AFI	C29
AGFI	C28
AGHI	A7B
AHI	A7A
ALFI	C2B
ALGFI	C2A
CFI	C2D
CGFI	C2C
CGHI	A7F
CHI	A7E
CLFI	C2F
CLGFI	C2E
IIHF	C08
IIHH	A50
IIHL	A51

Mnemonic	Opcode
IILF	C09
IILH	A52
IILL	A53
LGFI	C01
LGHI	A79
LHI	A78
LLIHF	C0E
LLIHH	A5C
LLIHL	A5D
LLILF	C0F
LLILH	A5E
LLILL	A5F
MGHI	A7C
MHI	A7D
NIHF	C0A

Mnemonic	Opcode
NIHH	A54
NIHL	A55
NILF	C0B
NILH	A56
NILL	A57
OIHF	C0C
OIHH	A58
OIHL	A59
OILF	C0D
OILH	A5A
OILL	A5B
SLFI	C25
SLGFI	C24
XIHF	C06
XILF	C07

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
A50	IIHH
A51	IIHL
A52	IILH
A53	IILL
A54	NIHH
A55	NIHL
A56	NILH
A57	NILL
A58	OIHH
A59	OIHL
A5A	OILH
A5B	OILL
A5C	LLIHH
A5D	LLIHL
A5E	LLILH

Opcode	Mnemonic
A5F	LLILL
A78	LHI
A79	LGHI
A7A	AHI
A7B	AGHI
A7C	MGHI
A7D	MHI
A7E	CHI
A7F	CGHI
C01	LGFI
C06	XIHF
C07	XILF
C08	IIHF
C09	IILF
C0A	NIHF

Opcode	Mnemonic
C0B	NILF
C0C	OIHF
C0D	OILF
C0E	LLIHF
C0F	LLILF
C24	SLGFI
C25	SLFI
C28	AGFI
C29	AFI
C2A	ALGFI
C2B	ALFI
C2C	CGFI
C2D	CFI
C2E	CLGFI
C2F	CLFI

In general, these immediate-operand instructions don't do anything you can't do with operands in memory. But on modern CPUs, they will execute much faster and will help reduce the size of your program.

Terms and Definitions

immediate operand

An operand contained in a field of the instruction itself.

Programming Problems

Problem 21.1.(2) Rewrite Problem 18.7 to generate a hexadecimal addition table, using immediate operands wherever possible.

Problem 21.2.(2) Rewrite Problem 18.8 to generate a hexadecimal multiplication table, using immediate operands wherever possible.

22. Branches, Loops, and Indexing

```
  2222222222    2222222222
 222222222222  222222222222
 22        22  22        22
           22            22
           22            22
          22            22
        22            22
      22            22
    22            22
  22            22
 222222222222  222222222222
 222222222222  222222222222
```

Programs often process data repetitively or iteratively under the control of a counter or some other condition. In this section we examine several instructions that simplify coding “loops”, sequences of instructions executed repeatedly.

First, we'll describe a newer form of branch instruction, the *relative-immediate* branch.

22.1. Branch Relative on Condition Instructions

The conditional relative branch instructions BRC and BRCL calculate their branch address by adding twice the immediate operand to the address of the branch instruction, as described in Section 20.1.3 on page 305. They have the formats shown in Tables 115 and 116.

A7	M_1	4	RI_2

Table 115. Format of the BRC instruction

RI-type branch instructions for which the value of the RI_2 operand lies in the range

$-2^{15} \le RI_2 \le 2^{15}-1$, or
$-32768 \le RI_2 \le 32767$

allow the *relative offset* to the branch target to lie as far as −65536 and +65534 bytes away.

C0	M_1	4	RI_2

Table 116. Format of the BRCL instruction

For RIL-type branch instructions the value of the RI_2 operand lies in the range

$-2^{31} \le RI_2 \le 2^{31}-1$, or
$-2147483648 \le RI_2 \le 2147483647$

This means the offset of the branch target can be more than 4 billion bytes away from the RIL-type instruction, in either direction.[134]

Relative branch instructions can help you reduce or even eliminate the need for base registers to address your instructions. We will describe conditional *relative* branches here, and examine other forms of relative branch shortly.

In almost every situation where you use an RX-type conditional branch (introduced in Section 15) you can replace it with a branch relative on condition instruction. For example, if you want to branch to the instruction named **Equal** if c(GR3)=c(GR12), you might have written a based-branch instruction like

```
         CR    3,12                Compare c(GR3) to c(GR12)
         BC    8,Equal             Branch if they're equal
```

or, you could use a relative branch by writing

```
         CR    3,12                Compare c(GR3) to c(GR12)
         BRC   8,Equal             Branch if they're equal
```

While this may seem extra effort for no obvious gain, the relative branch has one major advantage: the target of any relative branch can be very distant from the branch instruction, while a based branch target can be at most +4094 bytes from the address of the based branch. The only (and usually minor) disadvantage is that relative branch instructions can't be indexed.

Like the extended mnemonics shown in Table 61 on page 210, the Assembler supports a similar set of extended mnemonics for branch relative on condition instructions, listed in Table 117. Because the most-used forms of these extended mnemonics begin with the letter “J”, they are often called “Jump” instructions.

RI Mnemonic		**RIL Mnemonic**		**Mask**	**Meaning**
BRC	JC	BRCL	JLC	M_1	Conditional Branch
BRU	J	BRUL	JLU	15	Unconditional Branch
BRNO	JNO	BRNOL	JLNO	14	Branch if Not Ones (T) Branch if No Overflow (A)
BRNH	JNH	BRNHL	JLNH	13	Branch if Not High (C)
BRNP	JNP	BRNPL	JLNP	13	Branch if Not Plus (A)
BRNL	JNL	BRNLL	JLNL	11	Branch if Not Low (C)
BRNM	JNM	BRNML	JLNM	11	Branch if Not Minus (A) Branch if Not Mixed (T)
BRE	JE	BREL	JLE	8	Branch if Equal (C)
BRZ	JZ	BRZL	JLZ	8	Branch if Zero(s) (A,T)
BRNZ	JNZ	BRNZL	JLNZ	7	Branch if Not Zero (A,T)
BRNE	JNE	BRNEL	JLNE	7	Branch if Not Equal (C)
BRL	JL	BRLL	JLL	4	Branch if Low (C)
BRM	JM	BRML	JLM	4	Branch if Minus (A) Branch if Mixed (T)
BRH	JH	BRHL	JLH	2	Branch if High (C)
BRP	JP	BRPL	JLP	2	Branch if Plus (A)
BRO	JO	BROL	JLO	1	Branch if Ones (T) Branch if Overflow (A)
	JNOP		JLNOP	0	No Operation

Table 117. Extended branch relative on condition mnemonics and their branch mask values

[134] That ought to be enough for most programs. (But that's what they said at one time about 24-bit addressing.)

Note:

The letter L in these mnemonics sometimes means “Long” (as in JLU) and sometimes “Low” (as in JL).

The previous example could be rewritten as

```
CR    3,12              Compare c(GR3) to c(GR12)
JE    Equal             Branch if they're equal
```

and no base register is needed for the JE instruction.

The Assembler checks that the branch target is within the current control section, so that you won't accidentally branch to an instruction that isn't part of the program containing the branch. (Such a branch is allowed if the target is an external symbol; we'll discuss this case in Section 38.)

Using explicit offsets from the current instruction is generally a poor practice:

```
CR    3,12              Compare c(GR3) to c(GR12)
JE    *+10              Branch 10 bytes if they're equal
```

This will cause maintenance problems if another instruction is added or removed between the JE and whatever instruction is 10 bytes away. The assembler generates X'A784 0005', where the offset value 10 has been halved at assembly time so that the Effective Address at execution time will be correct.

An even poorer coding technique is writing the RI_2 operand explicitly:

```
CR    3,12              Compare c(GR3) to c(GR12)
JE    10                Branch 10(?) bytes if they're equal
```

The Assembler will issue a warning and then generate the instruction with the explicit absolute operand in the RI_2 field, X'A784 000A' so that the branch target is actually *20* bytes away!

Branch relative instructions can be very helpful in programs larger than 4K bytes, where using based branch instructions may require more than one base register to provide addressability. With relative branches, you can often reduce the number of “program base” registers (or even eliminate them entirely), freeing registers for other more productive uses.

Exercises

22.1.1.(1) The extended mnemonic for the “Long” relative branch instructions is formed by adding the letter L after the initial letter “J”. But the Long unconditional relative branch mnemonic is JLU, not JL. Why?

22.1.2.(1) Can you think of situations where JNOP and JLNOP will be useful?

22.1.3.(2)+ What machine instruction is generated by each of these statements?

```
(1)      J      *-10
(2)      J      *+2046
(3)      J      -40
(4)      JL     *+2
```

What will happen if the last of these is executed?

22.2. A Simple Example of a Loop

We will use variations on a simple example to illustrate some basic principles. Suppose a *string* — a one-dimensional array — of 80 bytes containing character data in the EBCDIC representation begins at **Str** and ends at **Str+79**. The character string could represent data read from an 80-character record.

We want to scan the string and replace all special (non-alphanumeric) characters by blanks. That is, any character with EBCDIC representation less than C'a' (X'81') should be replaced by C' ' (X'40').[135] Thus, letters and digits will be unchanged.

We begin with the example in Figure 167. It performs the required processing in a straightforward but perhaps clumsy way. This “problem” will be used for several more examples, so try to understand the basic idea here.

```
         SR    0,0                Characters are inserted into GR0
         LR    1,0                Character count in GR1, initially 0
         LA    2,C'a'             c(GR2) = X'00000081'
         LA    3,C' '             c(GR3) = X'00000040'
         LA    4,Str              First byte's address in GR4
GetChar  IC    0,0(,4)            Get a byte from the string
         CR    0,2                Compare to letter 'a'
         JNL   Okay               Branch if a letter or digit
         STC   3,0(,4)            Otherwise replace by a blank
Okay     LA    4,1(,4)            Increment character address by 1
         LA    1,1(,1)            Increase character count by 1
         C     1,=F'80'           Compare count to 80 (string length)
         JL    GetChar            Loop if less than 80 done so far
         - - -
Str      DC    CL80'String-to,be(Scanned+For*Special=Characters$#'
```

Figure 167. A simple loop to scan and replace characters

The character comparisons are made in the rightmost bytes of registers GR0 and GR2.[136] The address of the byte being examined is in GR4, and is incremented by 1 at each step, and was initialized to the address of the first character before entering the loop. The branch instruction at the end of the loop must branch if the contents of GR1 is *less* than 80, not if it is less than *or equal* to 80: otherwise, the final test would cause the byte at **Str+80** to be examined and possibly changed. The string ends at **Str+79**.

Exercises

22.2.1.(2)+ Show the result at **Str** after the program segment in Figure 167 completes execution.

22.2.2.(1) Revise the program in Figure 167 to use extended relative branch mnemonics and no literals.

22.3. Simple Tables and Array Indexing

The next version of this program uses the indexing capabilities of the IC and STC instructions.[137] Assume that the character string at **Str** has been defined as in Figure 167.

---

135 Table 13 on page 87 shows that in the EBCDIC character representation, all letters and numeric digits have encodings greater than X'80'.

136 LA (and not LHI) was used to load GR2 and GR3 with C'a' and C' '. LA and LHI are both very fast instructions; the CPU pays special attention to LA, because address generation is a fundamental operation. Basically, there's no detectable difference in speed.

137 Review Sections 5.3 and 9.5 for a quick summary of indexing.

```
         SR    0,0              Clear GR0 for character insertion
         SR    1,1              Initialize index to 0
         LA    3,C' '           c(GR3) = blank at right end
GetChar  IC    0,Str(1)         Get a character from string
         C     0,=A(C'a')       Compare to letter 'a'
         JNL   Okay             Jump if not less than X'81'
         STC   3,Str(1)         Replace by a blank
Okay     LA    1,1(,1)          Increment index by 1
         C     1,=F'80'         Compare to length of string
         JL    GetChar          Branch if not done
         - - -
```

Figure 168. A simple loop, using indexing

The byte being examined is now addressed using GR1 as an index register. The first time the IC instruction named **GetChar** is executed, the contents of GR1 is zero and the Effective Address generated will be the address of **Str**. On the last execution of the IC instruction, the contents of GR1 is 79, and the last byte of the string is inserted into GR0 for examination. Then, after the LA instruction named **Okay** is executed, the contents of GR1 is 80, the branching condition for the final JL instruction is not met, and control will pass to the following instruction.

A minor difference in this version is that the 32-bit word containing the EBCDIC representation of the letter **'a'** is now a word in memory, specified by the literal =A(C'a'), rather than in GR2 as before.[138]

Figure 169 illustrates another use of indexing. Three fullword integers stored beginning at **QQ** are added with tests for overflow. In this case, however, after the sum is complete, a branch to **NoErr** is made if no overflows occurred, to **Err1** if exactly one overflow occurred, and to **Err2** if two.

```
         SR    1,1              Set overflow count in GR1 to zero
         L     0,QQ             Load first integer into GR0
         A     0,QQ+4           Add second integer
         JNO   A1               Branch if no overflow
         LA    1,4(,1)          Indicate one overflow
A1       A     0,QQ+8           Add third integer
         JNO   A2               Branch if no overflow
         LA    1,4(,1)          Indicate an overflow
A2       B     BrTbl(1)         Indexed(!) branch into branch table
BrTbl    J     NoErr            0-overflow branch
         J     Err1             1-overflow branch
         J     Err2             2-overflow branch
```

Figure 169. Indexing into a branch table

When the instruction named **A2** is reached, GR1 contains the number of overflows multiplied by four. This is used as an index in computing the Effective Address of the BC instruction at **A2**, which will be **BrTbl**, **BrTbl+4**, or **BrTbl+8**. The appropriate branch instruction will then transfer control to the desired location. The symbol **BrTbl** need not be on a fullword boundary: the index in GR1 is incremented by 4 for each overflow to account for the length of the J instructions.

Branch tables provide a fast and efficient way to route control to different parts of a program.

Exercises

22.3.1.(2) A list of N halfword integers is stored beginning at **DATA** and the number N is a halfword integer at **NBR**. Write a code sequence that will store at the fullwords **POS**, **NEG**, and **NZT** respectively the sum of the positive terms, the sum of the negative terms, and the number of zero terms.

[138] While =F'129' and =A(X'81') would give identical results, using the fullword integer literal is a poor practice, because your reader can't tell that the literal is intended for use in a character comparison.

22.3.2.(1) What will happen if the conditional branch instruction named **A2** in Figure 169 is changed to a relative branch instruction?

22.3.3.(2)+ Revise Figure 168 on page 333 to use immediate operands to replace references to operands in memory.

22.4. Branch on Count Instructions

The branch on count instructions are shown in Table 118. None of them changes the CC setting.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
46	BCT	RX	Branch on Count (32)	06	BCTR	RR	Branch on Count Register (32)
E346	BCTG	RXY	Branch on Count (64)	B946	BCTGR	RRE	Branch on Count Register (64)
A76	BRCT	RI	Branch Relative on Count (32)	A77	BRCTG	RI	Branch Relative on Count (64)

Table 118. Branch on count instructions

Like the conditional relative branches, the Assembler provides extended mnemonics for the two branch relative on count instructions:

Instruction	Extended Mnemonic
BRCT	JCT
BRCTG	JCTG

Table 119. Extended mnemonics for branch relative on count instructions

The Branch on Count instructions simplify counting and branching operations like those in Figures 167 and 168 above. As with the BCR and BC instructions, the *branch address* is obtained either from R_2 for BCTR and BCTRG (unless the R_2 digit is zero, in which case no branch is ever taken); or from the Effective Address for BCT and BCTG.

The branch address is computed first. Then, the branching condition is determined by first arithmetically reducing the contents of R_1 by one, and branching *only* if the resulting contents of R_1 is not zero. That is, the branch does not occur only when the result is zero.

The CC is unchanged, and has no effect on the branching condition. An interruption condition is never recognized, even if an internal fixed-point overflow occurs (that is, if the new contents of R_1 "wraps around" from the largest negative number to the largest positive number).

We can rewrite our original example in Figure 167 on page 332 to use a JCT instruction, by stepping *backwards* along the string of characters starting at **Str+79** and ending at **Str**. This lets us use the same quantity both as an index and a counter.

```
         SR    0,0                 Clear GR0
         LA    1,80                Set GR1 to number of characters
         LA    2,C'a'              c(GR2) = letter 'a'
         LA    3,C' '              c(GR3) = blank
Next     IC    0,Str-1(1)          Get a character
         CR    2,0                 Compare 'a' to character
         JNH   Okay                Branch if 'a' is low or equal
         STC   3,Str-1(1)          Otherwise blank it out
Okay     JCT   1,Next              Count down by 1, jump if not 0
```

Figure 170. A backward loop to scan and replace characters

We used the implied address Str−1 in the second operand of the IC and STC instructions because the possible values in GR1 now run from 80 to 1, rather than from 0 to 79 as before. The range of values is different, but the direction of incrementation makes no difference in this example. This can be thought of as reflecting a difference in numbering the bytes in the string: if we number them from 0 to 79 they are addressed by writing the operand as Str(1); but if the bytes are numbered from 1 to 80, they must be addressed by writing the operand as Str−1(1).

On the final pass through the loop, the contents of GR1 will be 1; when the JCT instruction is executed, the contents of GR1 is reduced to zero, the branching condition is finally not met, and control passes to the next sequential instruction. We see an immediate gain in program efficiency over the example in Figure 168 on page 333: if we count the instructions inside the loop, we have reduced them from 7 to 5, and we would expect about the same reduction in processing time.

The Branch on Count instructions are especially useful when a predetermined number of loop iterations is needed, and no special attention must be paid to indexing quantities. The count of loop iterations is often set at execution time rather than at assembly time.

To illustrate several uses of these instructions, consider these examples taken from previous sections.

1. The word at **Nbr** contains a positive integer N; compute the sum of the cubes of the first N integers. (See Figure 125 on page 267.)

```
         L     4,Nbr            c(GR4) = index 'K', initially N
         SR    5,5              Initialize sum to zero
Next     LR    1,4              c(GR1) = K
         MR    0,1              K * K
         MR    0,4              K cubed
         AR    5,1              Add to sum
         JCT   4,Next           Decrease K by 1, and loop
         ST    5,Sum            Store sum
```

2. The halfword at **NN** contains a positive integer N; store at **NSq** the sum of the first N odd integers. (See Figure 82 on page 218.)

```
         SR    0,0              Clear sum to zero
         LH    1,NN             Get N from memory
Loop     LA    2,0(1,1)         (Count + Count) in GR2
         BCTR  2,0              (2 * Count) - 1
         AR    0,2              Add to sum
         JCT   1,Loop           Reduce count and branch
         ST    0,NSq            Store result
```

Figure 171. Calculate the sum of the first N odd integers

Because N is positive and at most 15 bits long, we can use the LA instruction to compute (N+N) in one step, since we know the result will fit in the rightmost 24 bits of GR2 for any addressing mode (so long as N is less than 2^{22}). The following BCTR instruction does not branch, because the R_2 digit is zero; its only effect is to reduce the contents of GR2 by one, as required. (The K-th odd integer is 2K−1.)

3. Find the two's complement of the double-length integer stored in the pair of words at **Arg**. (See Figure 91 on page 226.)

```
         LM    6,7,Arg          Double-length number in (GR6,GR7)
         LCR   6,6              Complement high-order part
         LCR   7,7              Complement low-order part
         JZ    XXX              Branch if carry out of GR7
         BCTR  6,0              Otherwise reduce c(GR6) by 1
XXX      STM   6,7,Arg          Store complemented result
```

This is identical to the example in Figure 91 on page 226 except that the BCTR instruction replaces

```
         SL    6,=F'1'
or
         AHI   6,-1
```

so the CC setting may be different when the STM is executed. The BCTR instruction with R_2=0 may be used this way anywhere in a program; it is shorter than subtracting a constant "1" from memory, but has the possible (minor) disadvantage that the CC is not set.[139]

As a further example of the BCT instruction, the program segment in Figure 172 stores the cubes of the integers from 1 to 10 in a table of ten successive fullwords starting at the word named **Cube**, but this time working backwards so the words are stored in descending order.

```
NCubes   Equ   10                  Number of table entries
         LA    4,NCubes            c(GR4) = number to be cubed
Mult     LR    3,4                 Move it to GR3
         MR    2,3                 Square it
         MR    2,4                 And cube it
         LR    1,4                 Set up index in RX
         SLL   1,2                 Multiply by 4 for word length
         ST    3,Cube-4(1)         Store in correct table position
         JCT   4,Mult              Branch back (NCubes-1) times
         - - -
Cube     DS    (NCubes)F           Space for 'NCubes' Words
```

Figure 172. Store the cubes of the first 10 integers

In this case we used the integer argument in GR4 to index the desired word in the table. Since the table entries are 4-byte words, the index must be multiplied by four for each item, so we use SLL to multiply. Because the first entry in the table corresponds to "1 cubed", the implied address of the ST instruction must be **Cube-4** so that the address of each entry will be calculated correctly.

Exercises

22.4.1.(2)+ In Figure 171 on page 335, show how you can eliminate one instruction from the body of the loop.

22.4.2.(1)+ In the BCT and BCTR instructions, what initial values of GR R_1 will cause a fixed-point overflow when the instruction is executed?

22.4.3.(2) A string of N bytes is stored beginning at **String**, and N is a halfword integer stored at **NN**. Store the string at **Gnirts** in reversed order.

22.4.4.(3) Suppose there is a nonnegative integer K whose value is stored in memory at the word integer **KK**. Starting at **Str** is a string of bytes whose bits are a random assortment of zeros and ones. Write a code sequence that will find the K-th one-bit in the string, and store its *bit offset* in the word at **BitOff**. For example, if the string starts with X'C607...', then if K=1 the bit offset is 0; if K=4 the bit offset is 6; and if K=6, the bit offset is 14.

22.4.5.(2)+ If b is the number of a register, what will happen if you execute these instructions?

```
BCT   b,0(,b)       or        BCT   b,0(b,0)
```

22.4.6.(3)+ A list of 100 fullword integers is stored beginning at the word named **IntList**. Write a code sequence that moves the integers into a list beginning at **NewList**, but do not move an item if it is identical to its predecessor. Store the number of items in the new list at **NumNews**. For example, if the first six values at **IntList** are 3, 5, 5, 5, 4, 3, then the list at **NewList** would begin with 3, 5, 4, 3.

[139] AHI and BCTR are both very fast, and AHI sets the condition code.

22.4.7.(2) The following code sequence is supposed to calculate the same sum of N odd integers as in Figure 171 on page 335. Why doesn't it? What does it calculate?

```
         LH    7,NN
         LA    1,1
XXX      LA    0,1(7,7)
         AR    1,0
         JCT   7,XXX
         ST    1,Nsq
```

22.4.8.(3) For the program below, determine first the machine code assembled for each of the instruction statements. Then when (at execution time) control reaches the SVC instruction, determine c(GR2). (Don't try to assemble and then execute the program, since the SVC will undoubtedly do something undesirable.)

```
Ex22_4_8 START 0
         USING *,8                 Establish addressability
         BASR  8,0                 Set base register
         LA    4,4                 Initialize counter
         LA    7,AA                Initialize address
         SR    2,2                 Set sum box to zero
Loop     NOPR  0                   Let the CPU catch its breath
         AH    2,0(,7)             Add a data item to the sum
         LA    7,2(0,7)            Increment address by 2
         BCT   4,Loop              Branch back if not done
         SVC   3                   Do something unforgettable
AA       DC    H'1,2,3,4,5,6,7,8,9'   Table of numbers
         END   Ex22_4_8
```

22.4.9.(4) Repeat Exercise 22.4.6, but this time the results at **NewList** may not contain any duplicate items. For example, if the six initial values at **IntList** are 2, 3, 2, 4, 2, 3, **NewList** would begin with the values 2, 3, 4.

22.4.10.(2) Write a sequence of instructions that will count the number of 1-bits in GG1 and leave the count in GG0. The original contents of GG1 need not be preserved.

22.4.11.(2)+ Repeat Exercise 17.2.6 using a BCT instruction to count the number of shifts.

22.4.12.(3)+ These instructions are intended to form the sums C(J)=A(J)+B(J) for values of J from 1 to 64. Show the generated object code, assuming that the PrintOut instruction generates exactly 32 bytes on the first available halfword boundary.

```
 Loc    Object Code         Assembler Language Statements
 ____   _____________                BASR  12,0
 ____   _____________                Using *,12
 ____   _____________                LA    3,64
 ____   _____________                LA    7,A
 ____   _____________                Using A,7
 ____   _____________       Loop     L     0,A
 ____   _____________                A     0,B
 ____   _____________                ST    0,C
 ____   _____________                BCT   3,Loop
 ____   _____________                Drop  7
 ____   _<32 bytes>__                PrintOut *
 ____   _____________       A        DS    64F
 ____   _____________       B        DS    64F
 ____   _____________       C        DS    64F
```

22.4.13.(2)+ In Exercise 22.4.12, the instructions don't perform the expected calculation. Explain what happens, and what needs to be fixed.

22.4.14.(3)+ An exercise required writing instructions to count the number of 1-bits in GR1 and leave the count in GR0. A student wrote:

```
         SR    0,0                 Set count to zero
         LA    2,32                Count 32 bits
Loop     SLL   1,1                 Shift a bit into sign position
         LTR   1,1                 Test sign bit
         BZ    Next                Branch if sign bit is zero
         LA    0,1(,0)             Add 1 to count of 1-bits
Next     BCT   2,Loop              Repeat for all 32 bits
```

The instructions didn't work. Find and fix the errors.

22.4.15.(3)+ Write instructions to count the number of 1-bits in GR1, leave the count in GR0, *and* leave GR1 unchanged without saving and then restoring its contents.

22.4.16.(2)+ State the cases in which each of the following five instruction sequences give different results, and explain the differences.

```
     (1) LCR  0,0        (2) X   0,=F'-1'    (3) X   0,=F'-1'
                             A   0,=F'1'         AL  0,=F'1'

     (4) BCTR  0,0       (5) S   0,=F'1'
         X     0,=F'-1'      X   0,=F'-1'
```

22.4.17.(2)+ Consider these three instructions:

```
         LCR   1,1
         BCTR  1,0
         LCR   1,1
```

1. What do these instructions do?
2. What instruction(s) do they imitate?
3. How do final Condition Code settings differ between these three instructions and the instruction(s) they imitate? For at least these initial values in GR1:

   ```
   (1) X'00000000'
   (2) X'00000001'
   (3) X'7FFFFFFF'
   (4) X'80000000'
   (5) X'FFFFFFFF'
   ```

   determine the resulting c(GR1) and the CC setting from executing the three instructions, and compare the results to what you get by executing the "imitated" instruction.

22.5. Looping in General

Most of these programming examples used loops to perform some iterative task, and the termination condition depended on a counting operation. More generally, many applications require that

- some quantity be established as an *index*
- whose value is changed regularly by an *increment*
- and is then compared to some *comparand*;
- a branch may then be made depending on some *condition* determined by the comparison.

These four terms — index, increment, comparand, and condition — will appear in several forms when we look at the branch on index instructions in Section 22.6.

The term “index” means the variable quantity that controls or determines completion of the loop; it may or may not be related to a value used as an index in an RX-type instruction (that is, specified by an index register specification digit).

If the increment is negative it might be more appropriate to call it a decrement. Rather than using special names to distinguish the sign of the increment, we will assume the increment can be either positive or negative.

Loops have many forms; here are two of the most common. The loops we have seen tested for loop completion at the end of the loop; this is called a “Do-Until” loop, because the loop is executed *until* the termination condition is reached. This is illustrated in Figure 173.

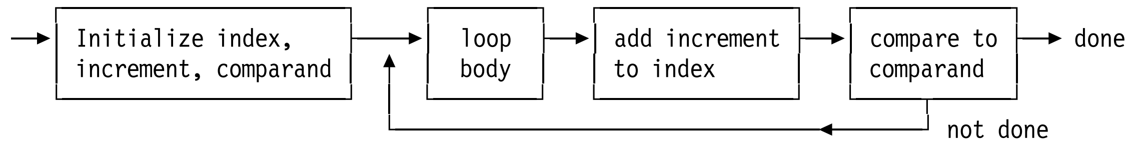


Figure 173. Sketch of a Do-Until loop

As this figure indicates, a Do-Until loop is always executed once.[140]

The other form is called a “Do-While” loop, because the loop is executed only *while* the termination condition has not been reached. This is illustrated in Figure 174.

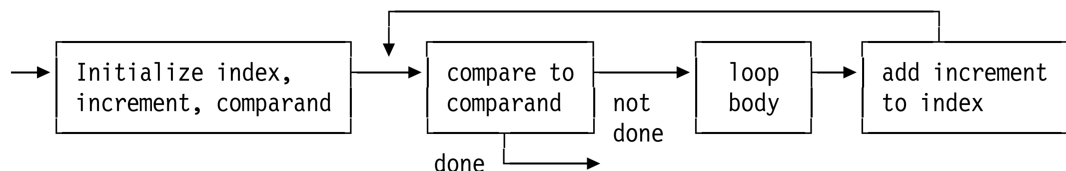


Figure 174. Sketch of a Do-While loop

For the Branch on Count instructions, the four loop-control items are all implied by the instruction: the index is in register R_1, the increment is -1, the comparand is zero, and the condition for branching is inequality. This rather limited set of possibilities may be sufficient for you to code your loop effectively.

Note!

The terminology for Do-While and Do-Until loops can be misleading. Such a loop is executed *until* the test condition becomes false, or only *while* the test condition remains true.

Figure 175 on page 340 shows another method to calculate a table of the first 10 cubes. The difference from Figure 172 on page 336 is that an address, rather than a subscripting index, is used as the varying quantity controlling execution of the loop.

140 For many years, this was the characteristic behavior of loops in the FORTRAN programming language.

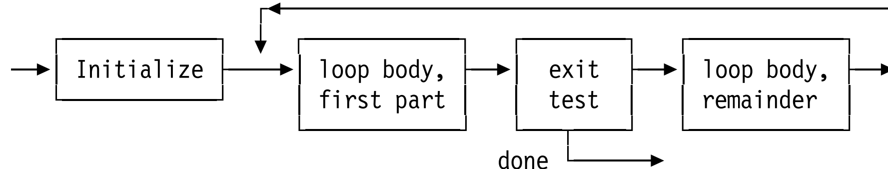
```
NCubes   Equ   10                  Number of table entries
         LA    1,Cube+0*4          Address of first table entry
         LA    2,Cube+(NCubes-1)*4  Address of last table entry
         LA    3,1                 c(GR3) = number to be cubed
Mult     LR    5,3                 Move multiplicand to GR5
         MR    4,3                 Square
         MR    4,3                 Cube
         ST    5,0(,1)             Store in table
         LA    3,1(,3)             Increment number to be cubed
         LA    1,4(,1)             Increment table address
         CR    1,2                 Compare to end address
         JNH   Mult                Jump back if not past end of table
         - - -
Cube     DS    (NCubes)F           Table of resulting values
```

Figure 175. Store the cubes of the first 10 integers in a different way

In this case an explicit address in the ST instruction is used, rather than an implied address as in Figure 172 on page 336. This means that the loop termination condition is determined from *address* arithmetic, not from tests on any of the quantities being calculated in the loop. It's often convenient to perform such addressing calculations explicitly, rather than rely on the Assembler to assign all bases and displacements. The “index” of the entries in the table can be thought of as running from 0 to (NCubes−1)*4 = 36 in steps of 4.

We used indexing in Figures 172 and 175 to compute a table of cubes. In Figure 172 on page 336, the “index” of the loop in GR4 is also used in GR1 to “index” the ST instruction; in Figure 175, the “index” of the loop is the address contained in GR1, but no RX-style “indexing” is done in any of the RX instructions.

Do-Until and Do-While loops are examples of “Structured Programming” forms, but other types of loop structures are often used. For example, you can test for a loop-exit condition in the body of the loop:



Exercises

22.5.1.(2) A table of N halfword integers is stored beginning at **HH**, and N is a halfword integer at **NHwds**. Store the integers into the table starting at **RR** in reverse order.

22.5.2.(2) Your solution to Exercise 22.4.9 will probably contain two loops. What are their types?

22.5.3.(1) What type of loop is illustrated in Figure 175?

22.6. Branch on Index Instructions

Because indexed loops are a key part of many programs, System z provides the Branch on Index High and Branch on Index Low or Equal instructions shown in Table 120 on page 341. They can greatly simplify coding of loops.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
86	BXH	RS	Branch on Index High (32)	EB44	BXHG	RSY	Branch on Index High (64)
87	BXLE	RS	Branch on Low or Equal (32)	EB45	BXLEG	RSY	Branch on Index Low or Equal (64)
84	BRXH	RSI	Branch Relative on Index High (32)	EC44	BRXHG	RIE	Branch Relative on Index High (64)
85	BRXLE	RSI	Branch Relative on Low or Equal (32)	EC45	BRXLG	RIE	Branch Relative on Index Low or Equal (64)

Table 120. Branch on index instructions

As there are no essential differences between BXH/BXLE and BXHG/BXLEG other than using 32-bit registers for the former and 64-bit registers for the latter, our examples will use the 32-bit forms. None of the instructions changes the CC setting.

As with Branch on Count, these instructions provide the three functions of incrementation, comparison, and conditional branching, but with much greater flexibility. BXH and BXLE are RS-type instructions requiring two register specification digits R_1 and R_3, as indicated in Tables 121 and 122.

opcode	R_1	R_3	B_2	D_2

Table 121. RS-type BXH and BXLE instructions

BXHG and BXLEG are RSY-type instructions that also require R_1 and R_3 operands:

opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

Table 122. RSY-type BXHG and BXLEG instructions

The relative-immediate forms of the branch on index instructions use two different instruction formats, RSI and RIE:

opcode	R_1	R_3	RI_2

Table 123. RSI-type BRXH and BRXLE instructions

opcode	R_1	R_3	RI_2		opcode

Table 124. RIE-type BRXHG and BRXLG instructions

Like the STM and LM instructions, the use of registers other than GR R_3 may be implied. First, note that *all* of the loop-control quantities (index, increment, and comparand) are carried in registers. The index is always in GR R_1, and the increment is always in GR R_3. The comparand is contained either in (GR R_3+1) (if R_3 is even), or in GR R_3 (if R_3 is odd).

Thus, if we write

```
BXLE   7,4,NEXT
```

then the index is in GR7, the increment is in GR4, and the comparand is in GR5. On the other hand, if we write

```
BXLE   7,5,NEXT
```

the index is again in GR7, but *both* the increment and the comparand are in GR5. Using an odd-numbered register for both the increment and the comparand will be discussed in Section 22.9.

We use a simple notational device to illustrate the fact that the comparand is always in an odd-numbered register: that is, if the R_3 operand is even, the comparand is in GR(R_3+1), and if the R_3 operand is odd, the comparand is in GR R_3. We write $R_3|1$ to mean that the register con-

taining the comparand is determined by ORing a low-order 1 bit into the R_3 digit. Thus, GR8|1 refers to GR9, and GR9|1 is the same as GR9.[141]

The operation of Branch on Index instructions, as sketched in Figure 176, is:

1. The sum of the index and increment is computed internally, and any overflow occurring in forming the sum is ignored.
2. The sum is then compared algebraically to the comparand. Whether or not the branching condition is met is noted: for "Branch on Index High" this means that the sum is algebraically greater than the comparand, and for "Branch on Index Low or Equal" that the sum is algebraically less than or equal to the comparand.
3. The sum then replaces the index, and the branch is taken if the branching condition is met.

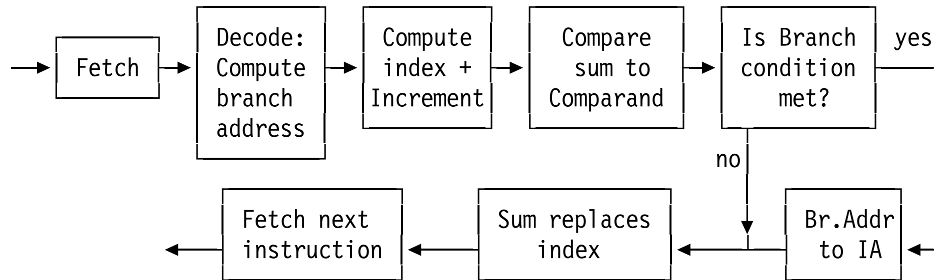


Figure 176. Operation of BXH and BXLE instructions

The branching condition is *not* reflected in the CC setting: neither of the "Branch on Index" instructions changes the CC.

Because the branch address is computed during the "Decode" portion of the instruction cycle *before* incrementation takes place, the Effective Address may not be as expected if the R_1 and B_2 digits are the same (unless both are zero, which is very unlikely.)

It's important to note that the comparison takes place *before* the sum replaces the index; we will see examples of situations where this is important. (Exercise 22.9.8 is recommended!)

Figure 177 shows another way to visualize the execution of BXH and BXLE.

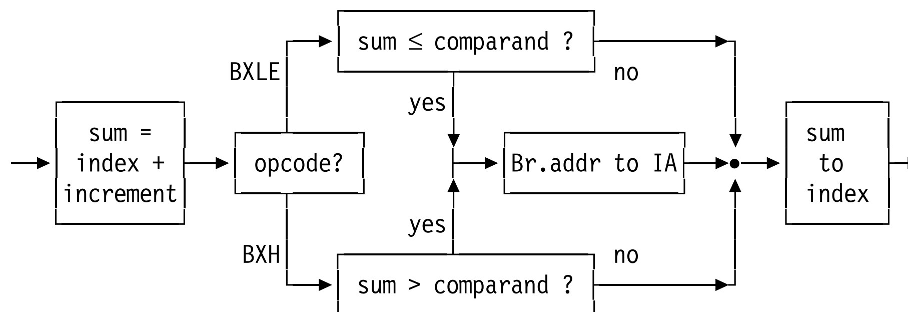


Figure 177. Operation of BXH and BXLE instructions

The Branch on Index instructions are powerful and useful, though they sometimes seem difficult. Normal uses require three general registers, of which two must be an even-odd register pair.

The placement of the comparand in R_3|1 rather than in R_3+1 (as would seem more useful and natural) is undoubtedly due to a design requirement for the original models of System/360: it was simpler to OR than to add a low-order one-bit to the register specification digit. Also, other

[141] We are using the PL/I-language notation for the logical "OR" operation, represented by the vertical-bar character "|".

double-length instructions such as M, D, and SLDA specify an even-numbered R_1 register, and the corresponding odd-numbered register may be "addressed" in the CPU by forcing a low-order one-bit into the register specification digit R_1.

Like the conditional relative branches, the Assembler provides extended mnemonics for the four branch relative on index instructions:

Instruction	Extended Mnemonic
BRXH	JXH
BRXHG	JXHG
BRXLE	JXLE
BRXLG	JXLEG

Table 125. Extended mnemonics for branch relative on index instructions

Exercises

22.6.1.(3) In the execution of the BXH and BXLE instructions, any overflow in forming the sum of the index and the increment is ignored. However, the comparison of the sum and the comparand requires an internal subtraction, in which an overflow *might* occur.

Make a table that includes all of the eight possible combinations of (1) BXH or BXLE, (2) sign of result of subtraction is + or −, and (3) an internal overflow did or did not occur during the subtraction. Determine for each of the eight combinations whether or not a branch will occur.

22.7. Examples Using BXLE

To illustrate BXH and BXLE, consider the example given in Figure 167 on page 332 in Section 22.2, where we want to replace non-alphanumeric characters by blanks. We'll rewrite the code sequence to use a BXLE instruction.

```
        LM    0,3,=F'0,0,1,79'   Preset registers GR0-GR3
*                                Chars inserted in GR0, index in GR1,
*                                increment in GR2, comparand in GR3.
        LM    4,5,=A(C'a',C' ')
*                                Letter 'a' in GR4, blank in GR5.
GetChar IC    0,Str(1)           Get a character from the string
        CR    0,4                Compare to letter 'a' in GR4
        BNL   Alpha              Branch if alphanumeric
        STC   5,Str(1)           Otherwise, store a blank
Alpha   BXLE  1,2,GetChar        Increment, test, and branch
        - - -
```

Figure 178. Replacing special characters with blanks, using BXLE

The values of the index run from 0 to 79; when control reaches the BXLE instruction, the increment (+1) in GR2 is added to c(GR1). Because GR2 is an even-numbered register, the sum is compared to the comparand in the next higher-numbered register, GR3. If the sum is less than or equal to 79, the branching condition is met, and control will be transferred to the instruction named **GetChar** after the sum is placed back into GR1. When control finally passes to the instruction following the BXLE, c(GR1) will be 80.

To give an example where BXLE appears in a more normal context, we will rewrite Figures 172 and 175 to compute a table of the cubes of the first 10 integers, stored starting at **Cube**.

```
NCubes   Equu  10                  Number of table entries
         LA    7,1                 Initial integer = 1
         SR    4,4                 Set index to zero
         LA    2,4                 Increment of +4 for indexing
         LA    3,4*(NCubes-1)      Comparand (=36) in GR3
Mult     LR    1,7                 N in GR1
         MR    0,1                 N * N
         MR    0,7                 N cubed
         ST    1,Cube(4)           Store in table
         AHI   7,1                 Increase N by 1
         BXLE  4,2,Mult            Increase index by 4 and loop
         - - -
Cube     DS    (NCubes)F           Space for table of cubes
```

Figure 179. Creating a table of cubed integers using BXLE

This segment uses fewer instructions inside the loop, at the expense of some extra instructions outside the loop: this is often a valuable technique, especially for loops executed many times. The following two code segments do the same calculation, but are set up slightly differently.

```
NCubes   Equ   10                  Number of table entries
         LA    7,1                 Initial value of N = 1
         LA    4,4                 Set increment in GR4 to 4
         LR    2,4                 Initial index in GR2 is 4
         LA    5,4*NCubes          Comparand in GR5 = 40
Mult     LR    1,7                 c(GR1) = N
         MR    0,1                 N squared
         MR    0,7                 N cubed
         ST    1,Cube-4(2)         Store in table
         AHI   7,1                 Increment N by 1
         BXLE  2,4,Mult            Count and loop
```

Figure 180. Creating a table of cubed integers using BXLE

In this example, the index runs from 4 to 40 in steps of 4, rather than from 0 to 36 as in Figure 179. There is no significant difference between the methods illustrated in Figures 179 and 180, except that the second *can* be simpler: since the integer N runs from 1 to 10 in steps of 1, the multiplication by 4 to account for the length of the fullword result makes it natural to have the index run from 4 to 40 in steps of 4. In Section 23 we will examine cases where such considerations are important, when we access tables of data stored in array form.

Another variation of this example is given in Figure 181, where the index and comparand quantities are addresses.

```
NCubes   Equ   10                  Number of cubes
         LA    4,Cube+0*4          Index set to initial table address
         LA    2,4                 Increment = 4 for fullwords
         LA    3,Cube+(NCubes-1)*4  Comparand = final table address
         LA    7,1                 Initial value of N = 1
Mult     LR    11,7                N
         MR    10,11               N * N
         MR    10,7                N * N * N
         ST    11,0(,4)            Store in table
         AHI   7,1                 Increment N by 1
         BXLE  4,2,Mult            Increment address by 4 and loop
```

Figure 181. Creating a table of cubed integers with addresses as controls

Exercises

22.7.1.(3)+ Examine these two instructions, and determine (1) whether the branch to **XX** will be taken, and (2) what will be the contents of GR3 after both instructions have been executed.

```
         LA    3,1
         BXLE  3,3,XX
```

Then, answer the same two questions, assuming that the second instruction is BXH instead.

22.7.2.(3)+ Suppose we execute the following two instructions:

```
         LA    3,3
         BXLE  3,3,*
Next     - - -
```

What will be in GR3 when the next instruction is executed? Make the same determination for BXH.

22.7.3.(3) A positive 64-bit dividend in registers GR6 and GR7 is divided by a positive divisor, using the D instruction. What will happen if the instruction following the divide is

```
         BXLE  6,7,WhatNext          ?
```

22.7.4.(2) In Figure 178 on page 343, combine the first two instructions into a single LM that uses a literal with an A-type constant. Then, initialize registers GR0 through GR5 using immediate operands. Including space required for the constants, which code sequence is shorter?

22.7.5.(4) Suppose registers GRx and GRy (where GRy is an odd-numbered register) contain nonnegative integers. A student claimed that we can leave in register GRx the sum of their contents modulo ($2^{31}-1$) with the following instruction pair:

```
         BXLE  x,y,*+8             Form c(GRx)+c(GRy)
         SL    x,=F'2147483647'    (2³¹−1)
```

Verify or disprove his claim.

22.7.6.(2) The following code sequence tries to find the leftmost 1-bit of the positive nonzero number in GR1, and put its bit number into GR0.

```
         SR    0,0                 Initialize bit position to 0
         LA    2,1                 Initialize BXLE increment
         LA    3,32                Initialize BXLE comparand
X        SLA   1,1                 Shift test word left once
         JM    Y                   Check for minus sign
         JXLE  0,2,X               Count up by 1 and loop
Y        - - -                     Rest of code
```

The program segment does not work correctly. Explain why not, and then correct it without increasing the number of instructions.

22.7.7.(3)+ By starting with a negative index value, it is possible to use a single register to hold the increment and comparand of a BXLE instruction. Rewrite the examples in Figures 179 through 181 to use this technique.

22.7.8.(2) Repeat Exercise 22.7.1, but replace the first instruction with the following:

```
         L     3,=F'1073741824'    (2³⁰)
```

Now, do the same again, replacing the LA by

```
         L     3,=F'-2147483647'   (-2³¹+1)
```

22.7.9.(3)+ If you execute this BXLE instruction:

```
         LM    1,3,=F'7,17,77'
         BXLE  1,2,*
```

How many times will the BXLE instruction be executed? How many times will it branch? What will be the sequence of values in GR1?

22.7.10.(2)+ Suppose A, B, and C are three positive integers used to initialize the index, increment, and comparand registers of a BXLE instruction that controls the iterations of a loop. How many times will the body of the loop be executed?

22.8. Examples Using BXH

To illustrate the use of the BXH instruction, Figures 179 and 181 will be rewritten so that the indexing runs in the opposite direction. First, we calculate the table of cubes using “normal” indexing.

```
         LA   7,10               Initial value of N
         LHI  8,-1               c(GR8) = -1 for incrementing N
         LA   4,40               Initial index = 40
         LHI  2,-4               Increment = -4
         SR   3,3                Comparand = 0
Mult     LR   1,7                N
         MR   0,7                N * N
         MR   0,7                N * N * N
         ST   1,Cube-4(4)        Store in table
         AR   7,8                Add -1 to N
         BXH  4,2,Mult           Count and loop
```

Figure 182. Creating a table of cubed integers using BXH

When the instruction following the BXH is reached, the index in GR4 will be zero.

We can use the value −4 for both the increment and the comparand *and* carry them in the same register, as in Figure 183.

```
         LA    7,10              Initial value of N is 10
         LA    4,36              Initial index = 36
         LHI   5,-4              Increment and comparand are -4
Mult     LR    1,7               N
         MR    0,7               N squared
         MR    0,7               N cubed
         ST    1,Cube(4)         Store in table
         BCTR  7,0               Decrease N by 1
         BXH   4,5,Mult          Count down and loop
```

Figure 183. Creating a table of cubed integers, using BXH in a special way

In this case the R_3 digit 5 is odd, so $R_3|1$ is the same as R_3; the BXH will increment the index in GR4 by −4, compare it to −4 (the comparand, also in GR5), and branch until the resulting sum becomes equal to −4, when control will pass to the following instruction.

Exercises

22.8.1.(3) Suppose we execute the instructions

```
SRL   1,1
BXH   1,1,*-4
```

Describe the behavior of this code segment as it depends on the initial contents of GR1. Then do the same, but with BXLE instead of BXH.

22.9. Specialized Uses of BXH and BXLE (*)

Some specialized uses of BXH and BXLE involve unusual combinations of register specification digits.

1. Suppose the contents of an odd-numbered register such as GR9 is zero. Then the instruction

```
XR    9,9                Set GR9 to zero
BXLE  4,9,XXX            Branch to XXX if c(GR4) is <= 0
```

   will branch to **XXX** only if the contents of GR4 is less than or equal to zero. Similarly,

```
BXH   4,9,YYY            Branch to YYY if c(GR4) is > 0
```

   would branch to **YYY** only if the contents of GR4 is greater than zero.

   Since BXH and BXLE neither set nor test the condition code, this technique can be used in situations where a condition code reflecting the state of the contents of GR4 is not available, the current CC setting must be undisturbed, or if we want to avoid using instructions such as LTR followed by a conditional branch.

2. Suppose we want to perform the inverse of the BCT instruction: that is, we want to increment the positive contents of a register by +1, and then branch. If we set c(GR7) to +1, and c(GR2) is greater than zero, then

```
LHI   7,1                Initialize GR7 to +1
BXH   2,7,XXX            Increment c(GR2), branch to XXX
```

   will branch to **XXX** after incrementing c(GR2) by 1, unless the sum overflows. (There will be no indication of the overflow in the CC setting!) Similarly, if there is a negative integer in GR2,

```
BXLE  2,7,YYY            Branch to YYY if c(GR2) not > +1
```

   will increment c(GR2) and branch to **YYY** so long as the resulting sum does not exceed +1.

3. If c(GR4) is +1, then the instruction

```
BXH   5,4,ZZZ
```

   will increment c(GR5) by 1 and then branch if the sum does not overflow. The index and comparand are in the same register: if the comparison was made *after* the sum was placed in GR5, equality would always be indicated, and the BXH would never branch.

Such special uses of the Branch on Index instructions are rare; they are used mostly in applications such as table searching and loop control. Try these exercises and the Programming Problems; you'll more fully appreciate the power of the branch on index instructions.

Exercises

22.9.1.(3) Suppose c(GR2)=5 and c(GR3)=73. What will be left in GR2 after executing these two instructions?

```
BXLE  2,2,*
SRL   2,1
```

More generally, if GR2 contains a small positive integer and GR3 contains a larger positive integer, what will be in GR2? Are there limits on the value of c(GR3)?

22.9.2.(3)+ What will be left in GR5 after executing these instructions?

```
LHI   5,1                Initialize GR5 to +1
BXLE  5,5,*              Do something interesting
```

Now, answer the same question for BXH.

22.9.3.(4) As in Exercise 22.7.6, the following code sequence tries to place in GR0 the number of the leftmost bit in the positive nonzero number in GR1. Prove that it works correctly. (Hint: consider the possible values of the two leftmost bits in GR1.)

```
         LA    0,1                 Initialize bit counter
         LR    2,0                 ... and bit count increment
         SR    3,3                 Zero comparand
Loop     BXH   1,1,ZBit            Skip if zero bit
         B     Done                Exit with bit number in GR0
ZBit     BXH   0,2,Loop            Increment bit count and try again
Done     - - -                     Bit number now in GR0
```

22.9.4.(4) What values in GR1 will cause the instruction

```
         BXH   1,1,Yes
```

to branch to the location named **Yes**?

22.9.5.(4) Repeat Exercise 22.9.4, but with a BXLE instruction.

22.9.6.(4) What values in GR0 and GR1 will cause the instruction

```
         BXH   0,1,Yes
```

to branch to the location named **Yes**?

22.9.7.(4) Repeat Exercise 22.9.6, but with a BXLE instruction.

22.9.8.(2)+ The operation of the Branch On Index instructions has often been described as follows:

1. The increment is added to the index, and the sum replaces the index.
2. The new index is compared to the comparand to determine the branch condition.

How is this description different from ours, and when and why is this description incorrect? Give an example showing how it would affect the actual operation of the Branch On Index instructions.

22.9.9.(4)+ This instruction sequence evaluates X**N (X^N) for 32-bit integer values of X and N. The base value X is in GR3, and the exponent value N is in GR0. Determine the algorithm used to evaluate the exponential; assume that no overflows occur.

```
         XR    1,1                 Clear GR1 to zero
         SRDL  0,1                 Shift low-order exponent bit to GR1
         BXH   1,1,OneBit          Branch if it was a 1-bit
ZeroBit  MR    2,3                 Was a 0-bit, square work value
         SRDL  0,1                 Shift another low-order bit for test
         BXLE  1,1,ZeroBit         Branch if it's zero to square again
OneBit   BXLE  0,1,Finished        Br if remaining exponent bits all 0
         LR    5,3                 More bits to do. Copy work value
Square   MR    4,5                 Square work value
         SRDL  0,1                 Move another bit for testing
         BXLE  1,1,TestMore        Branch if it's zero
         MR    2,5                 Otherwise multiply work into answer
TestMore BXH   0,1,Square          Branch if any 1-bits remaining
Finished - - -                     Result is in GR3
```

You will find it very instructive to follow this instruction sequence for several values of the exponent such as 1, 5, 8, 11, and 15.

22.10. Summary

The relative branch instructions discussed in this section are summarized in Table 126.

Operation	Relative-Immediate Operand Length	
	16 bits	32 bits
Branch on Condition (Relative)	BCR	BCRL

Table 126. Branch relative on condition instructions

The loop-control instructions discussed in this section are summarized in Table 127.

Operation	Register Length	
	32 bits	64 bits
Branch on Count (Register)	BCTR	BCTGR
Branch on Count (Indexed)	BCT	BCTG
Branch on Count (Relative)	BRCT	BRCTG
Branch on Index	BXH BXLE	BXHG BXLEG
Branch on Index (Relative)	BRXH BRXLE	BRXHG BRXLG

Table 127. Branch instructions for loop control

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
BCT	46
BCTG	E346
BCTGR	B946
BCTR	06
BRC	A74
BRCL	C04

Mnemonic	Opcode
BRCT	A76
BRCTG	A77
BRXH	84
BRXHG	EC44
BRXLE	85
BRXLG	EC45

Mnemonic	Opcode
BXH	86
BXHG	EB44
BXLE	87
BXLEG	EB45

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
06	BCTR
46	BCT
84	BRXH
85	BRXLE
86	BXH
87	BXLE

Opcode	Mnemonic
A74	BRC
A76	BRCT
A77	BRCTG
B946	BCTGR
C04	BRCL
E346	BCTG

Opcode	Mnemonic
EB44	BXHG
EB45	BXLEG
EC44	BRXHG
EC45	BRXLG

Terms and Definitions

comparand
A quantity to which an incremented index is compared to determine whether a loop should be repeated.

increment
A (normally) constant value used to update the value of an *index* for each iteration of a loop.

index
A varying quantity used to control each iteration of a loop.

$R_3|1$
A notation referring to the general register containing the comparand of a branch on index instruction. If the R_3 operand is even, $R_3|1$ is the next higher odd-numbered register; and if the R_3 operand is odd, $R_3|1$ is that odd-numbered register.

Programming Problems

Problem 22.1. Write a program to print a formatted hexadecimal multiplication table.

Problem 22.2. Each section of this text starts with large "block numbers" showing the section number. The blocks are 12 characters wide and 12 characters high.

Write a program that reads a single record containing up to 72 numeric digits, and print up to 10 "block number" digits at a time across the page, each separated from the preceding by 2 spaces. If more than 10 digits are provided on the input record, print 2 blank lines before each succeeding group. If a space appears in the input record, leave that 12-character position blank in the printed output. (Remember that the 12 blanks are separated from any preceding character by 2 spaces.)

Thus, if your input record contained only the three characters `'1 2'` (with a space between the two digits), your printed output would look like this; the bottom line is shown here only to help you understand the spacing.

```
     11                      2222222222
    111                     222222222222
   1111                     22        22
     11                               22
     11                               22
     11                              22
     11                            22
     11                          22
     11                        22
     11                      22
 1111111111                 222222222222
 1111111111                 222222222222
....+....1....+....2....+....3....+....4....+....5....+....6.... etc.
```

Some other sections are headed with "block letters". You will enjoy extending your program to handle letters as well as digits.*

* Such block-lettered pages were called "banner pages", and were often used to separate fan-folded printer outputs for one job from another.

Chapter VII: Bit and Character Data

```
VV        VV  IIIIIIIIII  IIIIIIIIII
VV        VV  IIIIIIIIII  IIIIIIIIII
VV        VV      II          II
VV        VV      II          II
VV        VV      II          II
VV        VV      II          II
VV        VV      II          II
 VV      VV       II          II
  VV    VV        II          II
   VV  VV         II          II
    VVVV      IIIIIIIIII  IIIIIIIIII
     VV       IIIIIIIIII  IIIIIIIIII
```

In previous chapters we discussed instructions that manipulated data in byte, halfword, word, and doubleword formats. The four sections of this chapter examine more basic System z instructions that work with individual bits and bytes, and with varying-length character strings.

- Section 23 shows how we can manipulate data consisting of single bytes and individual bits within a byte.
- Section 24 first introduces important concepts in using SS-type instructions. It then describes frequently-used instructions used to process data involving large or variable numbers of bytes, and introduces the powerful “Execute” instructions.
- Section 25 examines instructions that process very long byte strings, and strings containing a special character.
- Section 26 discusses other character representations such as ASCII, Unicode and other multiple-byte characters, and instructions to handle them.

23. Bit and Byte Data and Instructions

```
 2222222222    3333333333
222222222222  333333333333
22        22  33        33
          22            33
          22            33
         22          3333
       22            3333
     22                 33
   22                   33
 22           33        33
222222222222  333333333333
222222222222   3333333333
```

Instructions having an operand in the instruction itself are called *immediate* instructions: the operand is immediately available from the Instruction Register, rather than from another register or (more slowly) from memory. We saw examples of register-immediate operands in Section 21. Here, the target operand of an SI-type instruction is in *memory*, whereas the RI-type and RIL-type instructions in Section 21 refer to target operands in the general registers.

23.1. SI- and SIY-Type Instructions

SI- and SIY-type instructions let you manipulate byte and bit data. They use an 8-bit immediate operand contained in the second (I_2) byte of the instruction, in the two formats shown in Tables 128 and 129.

opcode	I_2	B_1	D_1

Table 128. SI-type instruction format

opcode	I_2	B_1	DL	DH	opcode

Table 129. SIY-type instruction format

The actions of the corresponding SI-type and SIY-type instructions are the same, so we'll describe only the SI forms. (Remember: the SIY-type instructions support a signed 20-bit displacement, while the SI-type instructions use an unsigned 12-bit displacement.)

The operand field is written as either

$D_1(B_1),I_2$ or S_1,I_2

showing the explicit and implied forms of address for the first operand.

The first operand of SI-type machine instruction statements typically refers to the name of a byte in memory. The second operand must be a nonnegative absolute expression of value less than 256, so that it will fit into the I_2 byte of the instruction.

Table 130 on page 353 describes the behavior of the instructions; the first operand is the single byte at the Effective Address.

Operation	Mnemonic	Action	CC set?
Move	MVI, MVIY	Operand 1 ← I_2	No
AND	NI, NIY	Operand 1 ← Operand 1 AND I_2	Yes
OR	OI, OIY	Operand 1 ← Operand 1 OR I_2	Yes
XOR	XI, XIY	Operand 1 ← Operand 1 XOR I_2	Yes
Compare	CLI, CLIY	Operand 1 Compared to I_2	Yes
Test Under Mask	TM, TMY	Test Selected Bits of Operand 1	Yes

Table 130. SI-type instruction actions

23.2. MVI Instructions

Table 131 lists the two Move Immediate instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
92	MVI	SI	Move Immediate	EB52	MVIY	SIY	Move Immediate

Table 131. Move Immediate instructions

MVI stores its I_2 operand into the byte at the Effective Address.

```
MVI   X,0             Set the byte at X to zero
MVI   X,255           Set the byte at X to all 1-bits
MVI   X,C'Y'          Store EBCDIC character 'Y' at X
MVI   X,C' '          Store EBCDIC blank at X
```

Figure 184. Examples of the MVI instruction

MVI is often used to initialize a byte whose bits will be used as bit flags, or to store a character. For example:

```
MVI   FlagByte,0      Set all flag bits to zero
MVI   CrrgCtrl,C'1'   Printer carriage control for new page
```

Exercises

23.2.1.(1) What do you expect will happen if you write these instructions?

```
MVI   0(4),B'000000000010101010'
MVI   0(4),B'000000000101010101'
MVI   0(4),-1
```

23.3. NI, OI, and XI Instructions

Table 132 summarizes these six Storage-Immediate instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
94	NI	SI	AND Immediate	EB54	NIY	SIY	AND Immediate
96	OI	SI	OR Immediate	EB56	OIY	SIY	OR Immediate
97	XI	SI	XOR Immediate	EB57	XIY	SIY	XOR Immediate

Table 132. Logical Storage-Immediate instructions

The CC settings after NI, OI, and XI are shown in Table 133 on page 354:

Operation	CC setting
AND OR XOR	**0:** all result bits are zero **1:** result bits are not all zero

Table 133. CC settings by SI-type logical instructions

The logical operations of the NI, OI, and XI instructions are between corresponding bits of the first and second operands, as we saw in Section 19. (You might want to review Figure 138 on page 289.)

```
(1)      NI   X,0              Same as 'MVI X,0' except CC set to 0
(2)      NI   X,253            Sets bit 6 at X to 0 (see below)
```

Figure 185. Examples of the NI instruction

Sometimes it is better to use other types of self-defining term for the second operand; example (2) could be written

```
         NI   X,B'11111101'
```

which more clearly shows that bit 6 will be zeroed.

```
(3)      OI   X,255            Same as 'MVI X,255' except CC set to 1
(4)      OI   X,B'00000010'    Sets bit 6 at X to 1

(5)      OI   LowerA,C' '      c(LowerA) now is C'A'
LowerA   DC   C'a'             Lower case letter 'a'
```

Figure 186. Examples of the OI instruction

```
         XI   X,B'0000010'     Inverts bit 6 at X
```

Figure 187. Example of the XI instruction

Exercises

23.3.1.(1) Example (5) in Figure 186 claims that the OI instruction changes `C'a'` to `C'A'`. Is this true? Why or why not?

23.3.2.(1) Write one instruction that will set the high-order and low-order bits of the byte at **`Flags`** to zero without affecting any of the other six bits.

23.3.3.(1) Write one instruction that will set the high-order and low-order bits of the byte at **`Flags`** to one without affecting any of the other six bits.

23.4. CLI Instructions

Table 134 shows the two Compare Immediate instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
95	CLI	SI	Compare Immediate	EB55	CLIY	SIY	Compare Immediate

Table 134. Compare Immediate instructions

The CLI instruction logically compares the byte in memory to the eight-bit I_2 operand as unsigned integers. The result is indicated by the CC setting, shown in Table 135 on page 355.

CC	Indication
0	Operand 1 = I_2
1	Operand 1 < I_2
2	Operand 1 > I_2

Table 135. CC settings after CLI instruction

You'll remember that the same settings are generated by the CL and CLR instructions, in Table 74 on page 232.

The following statements would result in the indicated CC settings. We use literals for the first operand so that both operand values are immediately visible.

```
CLI   =C'A',X'C1'        CC = 0:   c(Operand 1) = I2
CLI   =X'00',0           CC = 0:   c(Operand 1) = I2
CLI   =C' ',B'01000000'  CC = 0:   c(Operand 1) = I2
CLI   =X'1',X'2'         CC = 1:   c(Operand 1) < I2
CLI   =C'A',250          CC = 1:   c(Operand 1) < I2
CLI   =C'X',C'X'-1       CC = 2:   c(Operand 1) > I2
CLI   =X'1',X'0'         CC = 2:   c(Operand 1) > I2
```

Remember:

The *first* operand in a CLI comparison is always the byte in memory at the Effective Address.

We can rewrite the example in Figure 167 on page 332 (and its variations) to blank out the special characters in the string at **Str**, now using CLI and MVI instructions. We'll start at the right (high-addressed) end and scan from right to left.

```
        LA    1,L'Str             Initialize loop count to string len
Next    LA    2,Str-1(1)          Form character's indexed address
        CLI   0(2),C'a'           Compare addressed character with 'a'
        JNL   AlfaNum             Skip blanking if not less than 'a'
        MVI   0(2),C' '           Blank out if not alphanumeric
AlfaNum JCT   1,Next              Count down and loop
        - - -
Str     DC    CL80'String ...'
```

Figure 188. A simpler loop to scan and replace characters

Because SI-type instructions cannot be indexed, the LA instruction named **Next** generates the memory address for the character to be tested. The CLI instruction then compares the byte in memory at that address to the immediate operand `C'a'`. If the byte in memory contains a bit pattern with value greater than or equal to `C'a'`, the following JNL instruction will branch around the MVI instruction. If the branching condition is not met, the MVI stores an EBCDIC blank character into the character string. These two SI-type instructions have simplified the previous examples of the same process.

Exercises

23.4.1.(1)+ Suppose the length of a string of bytes starting at **Data** is not known, but we know that the end of the string is marked with a byte of all 1-bits. Write a code sequence which will leave the length of the string in GR1.

23.4.2.(1)+ In solving Exercise 23.4.1, a student wrote these instructions:

```
        SR    1,1                 Initialize index
Loop    LA    1,1(,1)             Increment by one
        CLI   Data-1(1),X'FF'     Test the byte
        BNE   Loop                Branch if not all 1-bits
```

Will this work?

23.4.3.(2)+ An 80-byte record starts at **Record**. Using CLI, find the address of the last non-blank character; store its address at **LastChAd** and store the length of the “initial” character string (from the first character to the last nonblank) at **DataLen**.

23.4.4.(2) Write an instruction that will set the Condition Code to 1 without changing any data or referencing any register, and without referencing any constants in storage.

23.4.5.(2) Write an instruction that will set the Condition Code to 2 without changing any data or referencing any register, and without referencing any constants in storage.

23.4.6.(1)+ A programmer tested a byte at Char for the lower case letter f, and wrote

```
         CLI   Char,f
```

He wasn't satisfied with the result; find two ways to help him.

23.4.7.(2) Write an instruction that will set the Condition Code to 0 without changing any data or referencing any register, and without referencing any constants in storage.

23.5. Test Under Mask Instructions

Table 136 shows the two Test Under Mask instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
91	TM	SI	Test Under Mask	EB51	TMY	SIY	Test Under Mask

Table 136. Storage-Immediate instructions

The Test Under Mask instruction is very useful in applications that examine bits. Because the CPU cannot directly address individual bits, data in bit form must be treated differently from data in byte or word form.

The I_2 (immediate) operand of a TM instruction is a *mask* indicating which bits of the addressed byte are examined: wherever a 1-bit appears in the mask, the corresponding bit position in the first operand is examined, and wherever a 0-bit appears in the mask, the corresponding bit of the memory operand is ignored. The result of the examination is indicated in the Condition Code, as shown in Table 137.

CC	Indication
0	Bits examined are all zero, or mask is zero
1	Bits examined are mixed zero and one
3	Bits examined are all one

Table 137. CC settings after TM instruction

If the I_2 mask is zero (meaning that *no* bits are tested), the CC is set to zero. The following examples illustrate uses of the TM instruction.

1. Branch to **Minus** if the fullword integer at **Num** is negative. (This technique can be used to avoid loading anything into a register.)

```
         TM    Num,X'80'          Test leftmost bit at Num
         JO    Minus              Branch if a 1-bit
```

2. Branch to **Even** if the fullword integer at **Num** is even.

```
         TM    Num+L'Num-1,1      Test rightmost bit of the word
         JZ    Even               Branch if bit is zero
```

3. Branch to **Mixed** if the bits of the byte at **BB** are not all zeros or all ones.

```
         TM    BB,255            Test all eight bits
         JM    Mixed             Branch if mixed zero and one
```

4. Branch to **Small** if the value of the halfword integer at **HNum** is between −512 and +511: that is, if the leftmost seven bits of the integer are all 0's or all 1's.

```
         TM    HNum,X'FE'        Test leftmost seven bits
         BC    9,Small           Branch if bits all zero or all one
```

The NI, OI, XI, and TM instructions let you set and test “on-off” and “yes-no” indicators in a program. For example, as in Figure 169 on page 333, suppose we wish to add the three fullword integers stored beginning at **Q**, and after all additions are done, branch to **NoErr** if no overflows occurred and to **Error** if one or more overflows occurred.

```
         NI    Flag,X'FE'        Set indicator bit for no overflows
         L     0,Q               Get first integer
         A     0,Q+4             Add second integer
         JNO   NextA             Branch if no overflow
         OI    Flag,1            Set overflow bit to 1 ('on')
NextA    A     0,Q+8             Add third integer
         JO    Error             Branch if overflow
         TM    Flag,1            Otherwise examine overflow bit
         JZ    NoErr             If bit was zero, no overflows
         JO    Error             If one, overflow occurred
         - - -
Flag     DS    X                 Overflow flag byte
Q        DS    3F                Integers to be added
```

Figure 189. Setting an overflow-indication flag bit

The OI instruction ORs a 1-bit into the rightmost bit position of the byte named **Flag**, setting it to a 1. Only the rightmost bit of the byte is modified, so the remaining seven bits could be used to indicate other conditions in the same program.

As another example of these instructions, suppose we have a list of N halfword integers stored at **List**, where the positive nonzero fullword integer N is stored at **NN**. We must add the elements of the list, except that alternate elements of the list are added twice. Whether the even-numbered or the odd-numbered elements are added twice is determined by the setting of the rightmost bit of the byte named **Switch**: if the bit is 1, the odd-numbered elements (beginning with the first) are added twice.

```
         LA    4,List            Initial list address in GR4
         L     3,NN              Number of elements in GR3
         SR    6,6               Initialize sum to zero
Load     LH    5,0(,4)           Get a halfword list element in GR5
         AR    6,5               Add to sum once
         TM    Switch,1          Test switch bit
         JZ    Once              Branch if zero, add only once
         AR    6,5               Add a second time
Once     LA    4,2(,4)           Increment list address by 2
         XI    Switch,1          Invert switch bit
         JCT   3,Load            Get next list element
         - - -
NN       DS    H                 Number of halfwords in the list
Switch   DC    B'0'              Byte with the 'switch' bit
```

Figure 190. Adding alternate list elements twice

Since the XOR of a 1-bit and any other bit inverts its value, the XI instruction alternately sets the switch bit to one and zero. The TM instruction examines only the rightmost bit of **Switch**, and the branching condition is met if the bit is zero.

Exercises

23.5.1.(2) In example 4 following Table 137 on page 356, show that if the leftmost seven bits of a halfword integer are all zeros or all ones, then the value of the integer lies between −512 and +511.

23.5.2.(3)+ Show that the operation of the TM instruction can be correctly described as follows:

1. Form internally the logical AND of the first operand and I_2. If the result is zero, set the CC to zero and go to the next instruction.
2. If the result of step 1 is nonzero, form internally the logical XOR of the result byte from step (1) and I_2. If the new result is zero, set the CC to 3 and go to the next instruction.
3. Otherwise set the CC to 1, and go to the next instruction.

23.5.3.(2) Write an instruction that will set the Condition Code to 3 without changing any data, and without referencing any register.

23.5.4.(2)+ A programmer needed to test the sign of a 4-byte binary integer stored at **BIN** without using any registers, and then branch to **POS** if the number was not negative. He wrote:

```
         TUM   BIN,80             Test Under Mask for sign bit
         BP    POS                Branch if nonnegative
```

Why didn't this work? Repair his instructions to work correctly.

23.5.5.(1) Use a TM instruction to set the Condition Code to zero without referencing any registers and without referencing any constants in storage.

23.5.6.(1)+ In example 3 of Section 23.5, can the extended mnemonic BNM be used to mean "Branch if Not Mixed"? Why?

23.6. Bit Data

The above examples illustrated SI-type instructions used mainly for control purposes. Another important application is to manipulate data in bit form, data that takes only two values. For example, suppose that the record of a person carrying automobile insurance requires the following "yes-no" information: (1) age less than 25? (2) male? (3) driver-training course completed? (4) married? (5) any previous claims? (6) assigned risk? Let the "yes" answers be represented by 1-bits in the byte named **Status**. Here are ways we could perform the given tasks.

1. The policy-holder has passed his 25th birthday.

```
Under25  Equ   B'10000000'         Define the young-person bit
         NI    Status,X'FF'-Under25     He's getting older now
```

2. The policy-holder has just married.

```
Married  Equ   B'00010000'         Define the married-person bit
         TM    Status,Married      Did he say he was already married?
         JO    Bigamy              (You never know!)
         OI    Status,Married      Indicate he's married now
```

3. The policy-holder has submitted a claim. If it is the first, branch to **Tsk**; otherwise, branch to **TskTsk**.

```
HasClaim Equ   B'00001000'         Define the made-a-claim bit
         TM    Status,HasClaim     Test if he claimed previously
         JO    TskTsk              Yes, must be accident-prone
         J     Tsk                 Accidents can happen to anyone
```

4. If the policy-holder is single, male, under age 25, *and* has not completed a driver-training course, branch to **HighCost**. As this example shows, you can test more than one bit with a single instruction:

```
Trained  Equ   B'00100000'        Define the driver-trained bit
Male     Equ   B'01000000'        Define the male-driver bit
         TM    Status,Married+Trained  Test 'Married' and 'Trained'
         JNZ   Next               Branch if both not zero
         TM    Status,Male+Under25     Test age and sex
         JO    HighCost           If young untrained male, branch
Next     - - -                    Rest of program
```

5. If the policy-holder is an assigned risk, indicate that he has previous claims if he also has no driver training.

```
Assigned Equ   B'00000100'        Define the assigned-risk bit
         TM    Status,Assigned    Check assignment status
         JZ    Next               Branch if not assigned
         TM    Status,Trained     Check driver training
         JO    Next               Branch if completed
         OI    Status,HasClaim    Otherwise set claim bit on
Next     - - -                    Rest of program
```

6. If the policy-holder is married, or has completed driver training, branch to **LowRisk**.

```
         TM    Status,Married+Trained  Check status
         JM    LowRisk            Branch if either but not both
```

These examples use EQU statements to assign symbolic names to values representing bits. Unfortunately, this is *not* the same as assigning a name to a bit itself; languages like PL/I have a BIT data type, but Assembler Language does not.[142]

Exercises

23.6.1.(2) Write instructions to format the bits in the byte at **BitData** as eight EBCDIC 0 and 1 characters starting at **BitChars**.

23.6.2.(1)+ In Example 6 of Section 23.6, can the extended mnemonic BNZ be used?

23.6.3.(1)+ Suppose we defined a bit with the statement

```
Over25   EQU   X'80'
```

How would you modify the statement in Example 1 of Section 23.6 to use this new definition?

23.7. Avoiding Bit-Naming Problems (*)

To illustrate a common problem using bit data, suppose we have defined two bytes containing flag bits, as follows:

```
Flag1    DS    X                  Define a byte containing flag bits
Bit0     Equ   X'80'              Name the value of bit 0 (leftmost)
Flag2    DS    X                  Another flag byte
Bit1     Equ   X'40'              And a value for bit 1 (next)
```

Under normal circumstances, we would refer to the bits with a code sequence like

```
         TM    Flag1,Bit0         Test a bit in flag byte
         JZ    SomeCode           Go do something if zero
```

The result of executing this TM instruction could easily be confused with

[142] You can use macro instructions to implement a bit-defining and bit-handling language that names bits and protects against referring to them accidentally.

```
         TM    Flag1,Bit1          Test bit 1 (in the wrong byte!)
         JZ    MoreCode            Branch if zero
or
         TM    Flag2,Bit0          Test bit 0 (in the wrong byte!)
         JZ    WhatCode            Branch if zero
```

If there is no way to force the "definitions" of **Bit0** and **Bit1** to be associated with their "owning" bytes, then if we use the wrong *byte* name we will test or manipulate the wrong *bits*. If we execute the instruction

```
         OI    Flag2,Bit0
```

we will set a bit in the wrong byte. Mistakes like this are not uncommon.

Here is a simple technique that avoids this naming problem: define **MyBit** and **HisBit** with the following statements:

```
MyBit    DS    0XL(X'80')          Define location and length attribute
         DS    X                   Reserve actual storage
HisBit   DS    0XL(X'40')          Define location, new length attribute
         DS    X                   Reserve actual storage
```

Figure 191. Defining bit names safely

The zero duplication factors mean that no storage will be reserved by the two bit-definition statements. The symbols **MyBit** and **HisBit** have the *value* attributes of the following byte, and their *length* attributes can be used to indicate which bit within each byte is desired. We then test the bit with an instruction sequence like

```
         TM    MyBit,L'MyBit       Test desired bit in correct byte
         JZ    YourCode            Branch if MyBit is zero
```

Figure 192. Using safely-defined bit names

and *no* reference will be made to (now nonexistent) symbols naming the bytes containing the **MyBit** and **HisBit** bits. Referring to **MyBit** only by its name and length attribute greatly reduces the chances of incorrectly referencing bit data.

Some IBM macros define bit names by their position in a byte:

```
Bit0     Equu  B'10000000'         Bit 0
Bit1     Equ   B'01000000'         Bit 1
Bit2     Equ   B'00100000'         Bit 2
Bit3     Equ   B'00010000'         Bit 3
Bit4     Equ   B'00001000'         Bit 4
Bit5     Equ   B'00000100'         Bit 5
Bit6     Equ   B'00000010'         Bit 6
Bit7     Equ   B'00000001'         Bit 7
```

If you use definitions like these to set a specific bit at a known position in a byte, the bit name indicates the bit's position. If, however, the bit is intended to have a meaning like "Initialization Complete" or "End of Input", it is much better practice give the bit a meaningful name:

```
InitDone Equ   B'00000001'         If 1, initialization completed
EndInput Equ   B'00000010'         If 1, no further input exists
```

Exercises

23.7.1.(2) Suppose the definition of MyBit in Figure 191 had been written

```
         DS    B
MyBit    Equ   *-1,X'80'
```

Would the instructions in Figure 192 work correctly? Why or why not?

23.7.2.(2)+ Using the bit-naming technique illustrated in Figure 191, define two bits named **BitA** and **BitB** in a single unnamed byte. Then, write code sequences to do the following:

1. Set BitA and BitB to zero.
2. Invert the value of BitB.
3. Branch to Both if BitA and BitB are both one.
4. Leave in GR0 the value of BitA+BitB (that is, a number which is 0, 1, or 2 depending on whether neither, either, or both bits are 1).

23.8. A Data Conversion Example

As a final example using SI-type instructions, suppose there is a fullword integer stored at NN that we want to convert to a character string of printable decimal digits. The sign of the number must precede the first digit; if the number is zero, the characters +0 should be placed at the right-hand end of the character string. Because a fullword integer can contain a value at most ten digits long in its decimal representation, we will reserve eleven bytes at **CharVal** for the result. We use the conversion method described in " 2.3. Converting Integers from One Base to Another (*)" on page 19.

The method shown here works, but is clumsy and complex. We will see when we examine packed decimal data in Section 30 that other instructions greatly simplify this task.

```
D        EQU   10                 Max number of digits
         LA    2,D                First, blank out result area
Blank    LA    3,CharVal-1(2)     Construct byte address
         MVI   0(3),C' '          Store blanks in first 'D' bytes
         JCT   2,Blank            Branch back (D-1) times
         LA    3,CharVal+D        Set up address of rightmost digit
         L     1,NN               Get number to be converted
         LPR   1,1                Take its magnitude
CnvtLoop SR    0,0                Clear high-order register
         D     0,=F'10'           Generate a digit by division
         STC   0,0(,3)            Store the remainder digit
         OI    0(3),X'F0'         Form correct EBCDIC representation
         BCTR  3,0                Move character pointer left by 1
         LTR   1,1                If quotient is zero, finished
         JP    CnvtLoop           If nonzero, generate more digits
         MVI   0(3),C'-'          Assume value was -, put sign
         TM    NN,X'80'           Check actual sign of argument
         JO    AllDone            Branch if it was indeed -
         MVI   0(3),C'+'          Sign is +, store character
AllDone  - - -                    Rest of program
CharVal  DS    CL(D+1)            Output character string, with sign
NN       DS    F                  Number to be converted
```

Figure 193. Converting a binary integer to characters

23.9. Instruction Modification (*)

In olden days, it was sometimes thought to be useful (or clever) to change the mask field of a conditional branch instruction, so that it alternately contained B'1111' and B'0000', causing an unconditional branch to alternate with a no-operation. The example in Figure 190 on page 357 might be rewritten as in Figure 194 on page 362 to use this technique.

```
         L     1,NN              Get number of elements to be added
         LA    0,2               Set up increment of 2 in GR0
         AR    1,1               2 * N
         SR    1,0               2 * (N-1) = comparand for BXLE
         SR    2,2               Initialize index in GR2 to zero
         SR    3,3               Same for sum, in GR3
         OI    Brnch+1,X'F0'     Set for single add on first pass
         TM    Switch,1          Check to see if setup is correct
         JZ    Add               Jump if branch setup is correct
         NI    Brnch+1,X'0F'     Otherwise set up to add twice
Add      AH    3,List(2)         Add a term from the list
Brnch    BC    0,FlipMask        Mask field alternated by XI inst'n
         AH    3,List(2)         Add again if required
FlipMask XI    Brnch+1,X'F0'     Invert branch mask bits again
         BXLE  2,0,Add           Count and loop
         ST    3,Result          Store answer
```

Figure 194. Adding alternate list elements twice, with program modification

The mask field of the BC instruction is addressed as **Brnch+1**, because **Brnch** is the name of the byte containing the operation code. Then, the instructions that manipulate the mask bits are written to leave unchanged the index register specification digit of the second byte of the instruction at **Brnch**, because we do not want to modify the index digit.

Modifying an instruction in memory is now considered a *terrible* programming practice, for these reasons:

1. The coding tends to be more difficult to understand, because you won't know with any certainty what is done by a given instruction if it could be modified by other parts of the program.
2. Debugging the program is more difficult, since it is usually easier to keep track of data (such as at **Switch** in Figure 190 on page 357) than parts of instructions. What you see in memory might not match your program listing. (It's no longer the program you wrote!)
3. If you must rewrite part of a program, it may be difficult to find all the instructions that modify or are modified by others.
4. If, as many programs are, the program must be *reenterable* (a property requiring no self-modification), such techniques are forbidden.
5. Modern processors assume that any instruction modifying memory is referring to data, so they prefetch large groups of instructions for faster decoding. If the CPU discovers that you have stored into the part of the program it prefetched, it must discard its initial analysis and re-fetch again. This can slow your program considerably.

Important Advice

Avoid self-modifying programs.

Most instruction modification needs are best handled by the Execute instruction, which we'll see in Section 24.11.

To show that the example in Figure 194 need not rely on program modification, the code segment in Figure 195 on page 363 does the same calculation more rapidly and safely.

Study the actions of the JXH and JXLE instructions carefully!

```
         L     1,NN              Set up JXLE comparand N in GR1
         BCTR  1,0               N-1
         ALR   1,1               2 * (N-1) = 2N-2 in GR1
         LA    0,2               Increment in GR0
         SR    3,3               Initialize sum to zero
         SR    2,2               Same for index
         TM    Switch,1          Test for first term adding twice
         JO    Twice             Branch if bit is 1, meaning yes
Once     AH    3,LIST(2)         Add a term once
         JXH   2,0,Done          Increment index, branch if done
Twice    AH    3,LIST(2)         Add a term
         AH    3,LIST(2)         ...twice
         JXLE  2,0,Once          Increment index and loop
Done     - - -                   Continuation of program
```

Figure 195. Adding alternate list elements twice, without program modification

Exercises

23.9.1.(3) The following fragment of code was discovered in a trash can. By examining the sequence of values contained in R4, determine what the code does.

```
         LA    6,2
         LA    4,5               Test number
VA       AR    4,6
         SLL   6,1
         XI    *-4,X'01'         Flip-flop
         - - -                   Some undecipherable material
         B     VA
```

23.9.2.(2) Show that the SI-type OI, NI, and XI instructions in Figure 194 on page 362 do not modify the index register specification digit of the instruction named **Brnch**.

23.9.3.(2) A widely used program (HASP) contained an instruction sequence like the following:

```
         OI    Flag+1,1          Set a flag bit
         - - -
Flag     TM    Flag+1,0          Test the flag byte
         BNZ   FlagSet           Branch if not zero
```

Elsewhere in the program, other instructions modified the byte at **Flag+1**. Why would anyone write a program this way?

23.10. Summary

The instructions we've discussed in this section are summarized in Table 138.

Function	Operand 1		Operand 2
	12-bit displacement	20-bit displacement	
Move Immediate	MVI	MVIY	I_2
AND Immediate	NI	NIY	I_2
OR Immediate	OI	OIY	I_2
XOR Immediate	XI	XIY	I_2
Compare Immediate	CLI	CLIY	I_2
Test Under Mask	TM	TMY	I_2

Table 138. Storage-Immediate instructions

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
CLI	95
CLIY	EB55
MVI	92
MVIY	EB52

Mnemonic	Opcode
NI	94
NIY	EB54
OI	96
OIY	EB56

Mnemonic	Opcode
TM	91
TMY	EB51
XI	97
XIY	EB57

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
91	TM
92	MVI
94	NI
95	CLI

Opcode	Mnemonic
96	OI
97	XI
EB51	TMY
EB52	MVIY

Opcode	Mnemonic
EB54	NIY
EB55	CLIY
EB56	OIY
EB57	XIY

Terms and Definitions

reenterable
A program is reenterable if

- Its execution can be suspended, then executed by other processes, and then resumed by the original process with correct behavior for all processes.
- It can be executed simultaneously by multiple processes, with correct behavior for all processes.

self-modification
A program modifies its instructions or constants. Considered a very poor programming practice with severe execution-time performance penalties, and forbidden if the program must be reenterable.[143]

[143] Technically, a self-modifying program can be reenterable if every execution instance makes exactly the same modifications. This is considered an even poorer practice.

24. Character Data and Basic Instructions

```
 2222222222           44
222222222222         444
22        22        4444
          22       44 44
          22      44  44
         22      44   44
       22       44444444444
     22        444444444444
   22                 44
 22                   44
222222222222          44
222222222222          44
```

The instructions we've seen thus far have involved at most one memory operand; now we'll investigate basic SS-type instructions that work with two operands in memory having variable lengths. We will also describe “Execute” instructions that help you handle varying-length data.

24.1. Basic SS-Type Instructions

We'll introduce some basic concepts using the instructions in Table 139.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
D2	MVC	SS	Move [Characters]	E8	MVCIN	SS	Move [Characters] Inverse
D4	NC	SS	AND [Characters]	D6	OC	SS	OR [Characters]
D7	XC	SS	XOR [Characters]	D5	CLC	SS	Compare Logical [Characters]
DC	TR	SS	Translate	DD	TRT	SS	Translate and Test
				D0	TRTR	SS	Translate and Test Reverse

Table 139. Basic character-handling instructions

The word “Characters” is enclosed in square brackets because the *z/Architecture Principles of Operation* description of those instructions omits that word from the name of the instruction, even though it's implied by the instruction mnemonics. While often used to manipulate character data, they simply process strings of *bytes*, whether or not they represent characters.

Because the lengths of the operands are not implied by the instruction (as we saw for instructions like L and LH), the number of bytes to be processed must be specified somehow. The instructions in Table 139 have the format illustrated in Table 140:

opcode	**L**	B_1	D_1	B_2	D_2

Table 140. Format of single-length SS-type instructions

These instructions are all 6 bytes long, have *two* Addressing Halfwords, and their second byte ("**L**") specifies the *machine length* or *Encoded Length*[144] of the operand or operands; we'll explain "Encoded Length" shortly.

The Assembler Language syntax of these instructions is shown in Figure 196:

```
mnemonic  D1(N,B1),D2(B2)
```

Figure 196. Assembler Language syntax of basic SS-type instructions

where **N**, the *Length Expression* (LE) (also known as the *program length*) is the number of bytes the instruction will process. (For some of these instructions, "at most" N bytes.)

The important difference between **N** and **L** is explained in Section 24.5 on page 370.

Except for TRT and TRTR, the only reference to or use of the general registers by the instructions in Table 139 on page 365 is for operand addressing.

The result of each operation is found in the *first* operand location, except for TRT, TRTR, and CLC, which modify no data in memory.

24.2. Operand Specifications and Explicit Lengths

As illustrated on page 114 in Section 9.9, you could write a typical SS-type instruction as

```
MVC   Field(5),Area
```

The operand field specifies three quantities: the implied addresses of the operands named **Field** and **Area**, and the number of bytes to be moved, 5.

Because the symbols **Field** and **Area** must be resolved into addressing halfwords, we must derive five operand-dependent quantities: the Encoded Length L and the base and displacement of the two addressing halfwords. The base and displacement of each addressing halfword is assigned by the Assembler from an implied address.

The number L in the Encoded Length byte generated by the Assembler is derived from the *Length Expression* (N) in your machine instruction statement. The Length Expression may also be explicit or implied; we'll discuss implied Length Expressions in Section 24.4.

You will remember from Section 8.5 on page 102 that machine instruction statement operands can take any of these three forms:

expr   *expr(expr)*   *expr(expr,expr)*

where the third format can sometimes be written *expr(,expr)*.

For most of the instructions we've seen so far, these formats are used for the first four instruction types shown in Table 141 on page 367, where S is our notation for an implied address, an absolute or relocatable expression. In the last row, we see that SS-type instructions introduce new possibilities:

[144] The Encoded Length byte is sometimes called the "machine length" or "Length Specification Byte".

Instruction Type	Operand Format			
	expr	*expr(expr)*	*expr(expr,expr)*	*expr(,expr)*
RR	register	invalid	invalid	invalid
RX	S	S(X)	D(X,B)	D(,B)
RS	S	D(B)	invalid	invalid
SI	S or immediate	D(B)	invalid	invalid
SS	S	S(N)	D(N,B)	D(,B)

Table 141. Instruction types and operand formats

In particular, notice that for SS-type instructions, the third operand format does *not* resolve to `D(X,B)`!

Suppose we want to move 23 bytes from the area of memory beginning at **AA** to the area beginning at **BB**. We could write

```
         MVC   BB(23),AA          Move 23 bytes from AA to BB
```

where the addresses of the two operands are implied. For SS-type instructions, the number in parentheses is not an index register specification, but an *explicit* Length Expression. Its value, 23, is the number N of bytes to be moved.

There are several ways to specify the Length Expression, as shown in Table 142. (Remember that "S_1" and "S_2" are our notations for the implied addresses of the first and second operands.)

Explicit Length
$S_1(N),S_2$
$D_1(N,B_1),S_2$
$S_1(N),D_2(B_2)$
$D_1(N,B_1),D_2(B_2)$

Table 142. SS-type instructions with explicit length

An *explicit* Length Expression is simply an expression you write in your machine instruction statement. Suppose we again want to move 23 bytes from **AA** to **BB** and that if GR9 is used as a base register, the displacements for **AA** to **BB** will be `X'125'` and `X'47D'` respectively. Then, Figure 197 shows how we could use any of the following four instructions, corresponding to the four operand formats in Table 142:

```
         MVC   BB(23),AA          S1(N),S2
         MVC   X'47D'(23,9),AA    D1(N,B1),S2
         MVC   BB(23),X'125'(9)   S1(N),D2(B2)
         MVC   1149(23,9),293(9)  D1(N,B1),D2(B2)
```

Figure 197. Examples of SS-type instruction operands

Equivalent decimal and hexadecimal self-defining terms are used for the displacements D_1 and D_2.

Exercises

24.2.1.(1) What is the difference between an implied address and an implicit address?

24.3. Symbol Length Attribute References

A Symbol Length Attribute Reference is written as the letter L followed by an apostrophe followed by a symbol, as in L'BB. It is an absolute term with value equal to the length attribute of the symbol. Because symbols can be defined in several ways, the following rules may be helpful:

1. If the symbol was defined in an EQU statement with * or a self-defining term in the operand field, its length attribute is one. However, as noted in Section 8.4 on page 100, if you specify a second operand in the EQU statement, that value will be used as the Length Attribute of the symbol. For example, if you define the symbol **XXX** in this EQU statement,

```
XXX      Equu   *,13
```

   then **XXX** will have the value of the current Location Counter, and length attribute 13.

2. The length attribute of a literal is defined; thus

```
         MVC   BB(L'=C'RAY'),=C'RAY'
```

   (while clumsy) is valid; it's better to define a constant named by a symbol, and then use the length attribute of the symbol:

```
         MVC   BB(L'RAY),RAY
         - - -
RAY      DC    C'RAY'
```

3. The length attribute of a Location Counter Reference (*) is the length of the machine instruction in which it appears. Thus MVC BB(L'*),AA assigns length attribute six, the length of the MVC instruction.

   - The length attribute of a symbol naming a macro instruction depends on the code it generates.

Exercises

24.3.1.(2) Can you find a way to specify an *implied* Length Expression whose value is zero?

24.3.2.(2) Is it possible to specify the length attribute of an expression using the L' notation?

24.4. Implied Lengths

If you don't specify an explicit length **N**, the assembler will derive an *implied length* from the first term in the first operand.

Implied Length
S_1,S_2
$D_1(,B_1),S_2$
$S_1,D_2(B_2)$
$D_1(,B_1),D_2(B_2)$

Table 143. SS-type instructions with implied length

Note that if the address of the first operand is specified explicitly and the Length Expression is implied, the comma following the left parenthesis is *very* important.

As a reminder, the words “explicit” and “implied” that we saw in Section 11.4 were used define constants with explicit and implied lengths:

```
IMPLIED  DC    F'8'                Implied length = 4 bytes
EXPLICIT DC    FL5'8'              Explicit length = 5 bytes
```

and the same words describe addresses:

```
ImplAddr L     0,=F'6'            Implied address, resolved by Assembler
ExplAddr L     0,X'D4'(0,7)       Explicit address, specified by you
```

In this section, we use the same words to describe Length Expressions: *you* can provide an explicit Length Expression, or you can let the Assembler *derive* the value of an implied Length Expression.

We usually don't want to have to specify an explicit Length Expression, particularly when the number of bytes should be apparent from the operands. For example, suppose the symbol **BB** is defined in a DS statement like this:

```
         MVC   BB,=120C' '        Set field at BB to blanks
         - - -
BB       DS    CL23               Field of length 23 bytes
```

Even though the second operand is 120 bytes long, if more than 23 bytes are moved by the MVC instruction, then the data or instructions following the byte at **BB+22** could be overwritten! Thus the length of the string of bytes to be moved should be determined from the first, or *receiving*, operand, rather than the second.

This is what the Assembler does. If no explicit Length Expression is given, the Length Attribute of the first operand is used as the value of the Length Expression: it is *implied* by the operand. In this example, the length attribute of the symbol **BB** is 23.

If the first operand is an expression rather than a single term, the length attribute is that of the leftmost term in the expression. Thus, with **BB** defined as above, if we write

```
          MVC   BB-4+X'5'-1,=120C' '
```

then the length attribute of the first operand is 23, but if we write

```
          MVC   X'5'+BB-5,=120C' '
```

the length attribute of the first operand is 1 because the length attribute of a self-defining term is always 1 (see Section 7.6).

Now, suppose we want to use an implied length, but with an *explicit* first operand address. The value of the Length Expression cannot be immediately associated with a symbol that names an area of the program using its length attribute. Unlike the examples in Figure 197 on page 367, knowing the base and displacement of the symbol **BB** (9 and X'47D') does not necessarily give the correct Length Expression when an implied length must be found. If an explicit base and displacement are given, the value of the Length Expression is the length attribute of the displacement expression. Thus

```
         MVC   X'47D'(,9),AA
```

specifies an implied length of 1 rather than 23, because X'47D' is a self-defining term. Using an explicit address and an implied length is very rare; it's much better to use a Length Attribute Reference.

You can specify an explicit base and displacement, and still use an implied length. We could have written

```
         MVC   BB-BB+X'47D'(,9),AA
```

and the length attribute of the displacement expression is the length attribute of **BB**, but this is cumbersome and confusing. We can rewrite this example to use an explicit base and displacement, in the improved form

```
         MVC   X'47D'(L'BB,9),AA  Length Expression = L'BB
```

Figure 198. SS-type instruction using a Length Attribute reference

It is almost always better to use a *Symbol Length Attribute Reference*, which is one of the *terms* we described when examining *expressions* in Section 8.1.

These rules are summarized in Table 144 on page 370. The last column shows how the Length Expression is determined for the four possible forms of the first operand.

First operand form	Address specification	Length Expression	Length used
S_1	implied	implied	$L'S_1$
$S_1(N)$	implied	explicit	N
$D_1(,B_1)$	explicit	implied	$L'D_1$
$D_1(N,B_1)$	explicit	explicit	N

Table 144. Determining the Length Specification Byte

Advice: Use Implied Lengths

Wherever possible, use implied lengths and let the Assembler derive the Length Expression for you. If the length of a data field changes, the Assembler will recalculate the Length Expression; this is safer and much more convenient than updating explicit Length Expressions manually.

Exercises

24.4.1.(1) How many bytes will be moved by these instructions? What values will you find in the first operand fields?

```
(1)      MVC   A,=X'01020304050'
A        DS    H

(2)      MVC   B,=CL6'ABCDEF'
B        DS    BL4

(3)      MVC   C(3),=F'2'
```

24.4.2.(2)+ What are the formats of each of these possible operands when used as (a) the first operand, and (b) the second operand of an MVC instruction?

(1) `7(4)` (2) `24(6,12)` (3) `A(B)` (4) `5(,1)`

24.4.3.(2)+ The paragraph following Table 141 on page 367 says that the comma following the left parenthesis is very important in some situations. Why?

24.5. The Encoded Length “L” and Program Length “N”

Now that we know how to write SS-type instruction statements with any Length Expression we need, we will review what actually goes into the machine language instruction. As we noted in Section 24.2, the value of the Encoded Length isn't necessarily the same as the value of the Length Expression. Here's why.

Why do we use **L** in the object code format, but **N** in the machine instruction statement format? They are different: ***L*** *is one less than* ***N*** (unless **N** is zero, in which case **L** is also zero). This is important!

There are good reasons for this difference:

- programmers want to specify **N**, the true number of bytes involved;
- the CPU must sometimes know the address of the rightmost byte of an operand; that address is the operand's Effective Address (its starting address) plus **L**;
- it makes no sense to operate on zero bytes (that's what NOP instructions are for!).
- the Execute instructions in Section 24.11 will show why instructions with zero length bytes are very useful.

When you code a value N in a machine instruction statement, the Assembler converts it to the correct value of L in the generated object code.

Because the Length Specification Byte is a single byte, it can have any value between 0 and 255; these actually specify operand lengths between 1 and 256. This is due to two factors:

- Every SS-type instruction always operates on at least one byte.
- All the instructions in Table 139 on page 365 *except* MVCIN and TRTR process data from left to right in order of increasing addresses.
  - For left-to-right instructions, the CPU must calculate the address of the last byte of each operand to check for possible memory-access violations.
  - Similarly, MVCIN, TRTR, and some other instructions process data from right to left, in order of *decreasing* addresses, starting at the rightmost byte.

  In both cases, the CPU must compute the addresses of the leftmost and rightmost bytes of the operand. It is simplest to locate the rightmost byte by adding the Encoded Length **L** to the effective address of the operand; if there are N bytes in a string starting at address A, its rightmost byte is at address A+N−1 = A+L.

That's why the Encoded Length has the value of the Length Expression minus 1.

Important Notation Difference!

The *z/Architecture Principles of Operation* illustrates SS-type instructions like MVC this way, where the two uses of **L** can be very confusing:

```
CLC  D1(L,B1),D2(B2)                          [SS]
```

D5	**L**	B1	D1	B2	D2

Both the Assembler Language syntax with operands $D_1(L,B_1),D_2(B_2)$ *and* the format of the assembled instruction use the ***same*** letter "**L**" to indicate the operand length! But the two numbers are *not* the same: the first "**L**" in the Assembler Language statement is the Length Expression (that we call **N**, the true number of bytes to process), while the second "**L**" in the assembled instruction is the Encoded Length, one less than the true length!

When you refer to the *z/Architecture Principles of Operation*, be *very* careful to distinguish the two uses of **L**.

We usually don't care about this distinction because we let the Assembler determine the needed quantities from the operands of the instruction statement. However, at *execution* time we may need to calculate the number of bytes to be manipulated, so it's important to understand this relationship between the Encoded Length and the actual number of bytes involved. An illustration (showing a bad way to do this) is given in Example 4 of Section 24.6; the *right* way to do this is discussed in Section 24.11.

Thus, the Encoded Length is a number *one less* than the value of the Length Expression, unless an explicit length of zero is given, in which case the Encoded Length is also zero.

Encoded Length

The Encoded Length is one less than the Length Expression, unless the Length Expression is zero.

The instructions in Figure 199 on page 372 would be assembled as indicated, assuming the same displacements for the symbols **AA** and **BB** relative to the base address in GR9, as in Section 24.2.

```
*        Instruction            Assembled form
*
         MVC   BB(23),AA        D216 947D 9125   LE=23
         MVC   BB(1),AA         D200 947D 9125   LE=1
         MVC   BB(0),AA         D200 947D 9125   LE=0
         MVC   0(L'*,0),29(12)  D205 0000 C01D   LE=6   (MVC's length!)
         MVC   15(L'BB-4,3),BB  D212 300F 947D   LE=19
         MVC   BB,AA            D216 947D 9125   LE=23
         MVC   H(L'H,H),H       D200 8008 0008   LE=1
         MVC   H(H,H),H(H)      D207 8008 8008   LE=8
         MVC   H+BB-AA(,9),AA   D200 9360 9125   LE=1
         MVC   T,BB-4           D216 947D 9479   LE=23
         MVC   BB-AA+4(9),AA    D208 035C 9125   LE=9
         - - -
BB       DS    CL23
T        EQU   BB               Length attribute of T = 23
H        EQU   8                Self-defining term, Len Attr = 1
```

Figure 199. Examples of Length Specification Bytes

Possible Confusion?

Sometimes people call "the value of the Length Expression" simply "the length". This can be confusing if "the length" is understood to mean the contents of the Encoded Length, which is sometimes called the "machine length". That is:

- **You** provide implicitly or explicitly a Length Expression (a "symbolic length" or "program length").
- The **Assembler** generates the Encoded Length (the "machine length").

24.6. The MVC and MVCIN Instructions

24.6.1. MVC: Move Characters

MVC moves the specified number of bytes starting at the second operand address to an area starting at the first operand address. There are no restrictions on overlapping areas, so you can do things like propagate a character through an area, or shift the bytes in an area. We need only remember that almost all SS-type instructions are executed in such a way that each byte is stored before the next source byte is accessed.

Figure 200 shows how MVC can be "emulated" by other instructions. Remember that the length expression LE is not the Length Specification Byte of a "real" MVC.

```
*Emulate MVC   BB(LE),AA        Moves LE bytes from AA to BB
         LA    1,BB             Address of first operand
         LA    2,AA             Address of second operand
         LA    0,LE             Length Expression (N)
         LTR   0,0              Check for zero
         JNZ   MoveByte         Nonzero, OK to move
         LA    0,1              LE=0 means move one byte
MoveByte IC    3,0(,2)          Get a second-operand byte
         STC   3,0(,1)          Store at first operand
         AHI   1,1              Increment first operand address
         AHI   2,1              Increment second operand address
         JCT   0,MoveByte       Repeat until LE bytes moved
```

Figure 200. Emulated operation of MVC instruction

Because MVC has no restrictions on operand overlap, the "byte at a time" emulation in Figure 200 is "faithful" to the execution of MVC. Of course, the MVC instruction doesn't modify any registers this way.

Here are some examples using MVC instructions.

1. Set the 120-byte area beginning at **Line** to blanks.

```
         MVI   Line,C' '          Store EBCDIC blank at 'Line'
         MVC   Line+1(119),Line   Propagate through rest of area
```

   This is sometimes called a "ripple" move. It requires less storage space than

```
         MVC   Line(120),=120C' '
```

   because extra space is required for the literal string of 120 blanks.

   Another way to set the 120-byte area at **Line** to blanks:

```
         MVC   Line,Line-1        Requires carefully-ordered DC's
         - - -
Blank    DC    C' '               Single blank
Line     DS    CL120              Immediately follows the blank
```

2. Shift the 80-byte character string beginning at **Str** to the left by two character positions, leaving blanks in the vacated positions.

```
         MVC   Str(78),Str+2      Move left by 2 bytes
         MVC   Str+78(2),=C'  '   Two blanks at end
```

3. Exchange the contents of the halfword integers at A and B.

```
         MVC   Temp,A             Move A to temporary location
         MVC   A,B                Move B to A
         MVC   B,Temp             Move old c(A) from Temp to B
         - - -
Temp     DS    XL2
A        DS    H
B        DS    H
```

4. GR8 and GR9 contain respectively the address and length of a message whose length is positive and less than 120 characters. Move the message to the area named **Line**.

```
         BCTR  9,0                Decrease length by 1
         STC   9,MVC+1            Store in length byte of MVC (??)
MVC      MVC   Line(0),0(8)       Move correct number of bytes
```

   The BCTR reduces the character count in GR9 from its "true" value to the "machine length" value required by the MVC: one less than the actual number of bytes to be moved. This is a *terrible* way to do this, because it requires instruction modification (discussed in Section 23.9). A a better way to do this uses the Execute instruction, which we'll see in Section 24.11.

24.6.2. MVCIN: Move Characters Inverse

MVCIN was implemented to support languages written from right to left. It moves characters the same way as MVC, but in reverse order. That is, the bytes of the second operand are fetched in *right-to-left* order and are stored at the first operand in *left-to-right* order. The second operand of the machine instruction statement must address the rightmost byte of the string to be moved. For example:

```
         MVCIN CRev,Chars+L'Chars-1     Move reversed Chars to CRev
         - - -
Chars    DC    C'12345'           Data to be moved
CRev     DS    CL(L'Chars)        Moved data = C'54321'
```

Figure 201. Example of Move Inverse instruction

As Figure 200 on page 372 does for MVC, Figure 202 shows an emulation of MVCIN. The emulation uses GR3 both as an index and as a count of the number of bytes to move.

```
*Emulate MVCIN BB(LE),AA+L'AA-1   Move inverse: LE bytes from AA to BB
         LA    3,LE               Number of characters to move in GR3
         LTR   3,3                Check for LE = 0
         JNZ   LENotZro           Skip if greater than zero
         LA    3,1                LE = 1 moves one byte
LENotZro LA    1,BB               Address of first operand
         LA    2,AA-1             A(2nd operand's leftmost byte)-1
Insert   IC    0,0(3,2)           Insert a byte from right end of AA
         STC   0,0(,1)            Store at left end of BB
         AHI   1,1                Increment first operand address
         JCT   3,Insert           Reduce byte count by one and loop
```

Figure 202. Emulated operation of MVCIN instruction

The emulated addressing seems to reference the byte preceding the leftmost byte of the second operand. However, the indexed IC instruction will start by inserting the rightmost byte and will end with the byte at AA, because GR3 actually contains the Length Expression, not the Length Specification Byte.

Unlike MVC, the *z/Architecture Principles of Operation* does not guarantee "byte-by-byte" operation for MVCIN, so if the operands overlap by more than one byte, the results may be unpredictable.

24.6.3. MVCOS: Move Characters With Optional Specifications (*)

The specialized MVCOS instruction can simplify character moves.

Op	**Mnem**	**Type**	**Instruction**
C80	MVCOS	SSF	Move [Characters] with Optional Specifications

Table 145. MVCOS instruction

The SSF instruction format illustrated in Table 146 is used for relatively few instructions.

opcode	R_3	op	B_1	D_1	B_2	D_2

Table 146. SSF instruction format used for the MVCOS instruction

Unlike the previous SS-type instructions, you specify the *true* length to be moved in a register, set GR0 to zero,[145] and the instruction will move up to 4096 bytes at a time. Its syntax is

```
MVCOS  D1(B1),D2(B2),R3
```

where data is moved from the second operand to the first, and the number of bytes to move is placed in the R_3 operand register (which of course must not be GR0!). The number of bytes actually moved is the number in the R_3 register or 4096, whichever is less. The CC is set to 0 if all bytes have been moved, or to 3 if more than 4096 bytes were specified.

For example, suppose we want to move 10000 bytes from **Here** to **There**:

[145] If nonzero bits appear in GR0, your program may cause a privileged-operation exception.

```
         LA    7,There            Target address
         LA    4,Here             Source address
         LHI   12,10000           Number of bytes to move
         XGR   0,0                Set GR0 to zero (important!)
Mover    MVCOS 0(7),0(4),12       Move up to 4096 bytes
         JZ    Done               Branch if all bytes moved
         AHI   7,4096             Update target address
         AHI   4,4096             Update source address
         AHI   12,-4096           Reduce remaining count
         J     Mover              Repeat for more bytes
Done     - - -
```

Figure 203. Example of MVCOS instruction

This example illustrates these important points:

- The source and target addresses, and the remaining byte count, are *not* updated by MVCOS: you must do that.
- Setting GR0 to zero is *very* important: MVCOS is a semi-privileged instruction, and any nonzero bits in GR0 may cause a program interruption (unless your program is executing in Supervisor State).

The convenience of MVCOS compared to MVC or MVCL may be outweighed by its slightly slower performance.

Exercises

24.6.1.(2) Suppose MVCIN used predictable “byte-by-byte” steps for any degree of operand overlap. What result would appear in Figure 201 on page 373 if the instruction was

```
         MVCIN Chars,Chars+L'Chars-1   ?
```

24.6.2.(2) The character string at **Message** has three segments, as defined by these statements:

```
Message  DS    0C
Prefix   DS    CL43
Insert   DS    CL29
Suffix   DS    CL67
```

Write instructions that will move the strings at **PText**, **IText**, and **SText** into these fields, but the string at **IText** must be moved to **Insert** in reverse order.

24.6.3.(1) What will be in the character string at **Result** after executing these instructions?

```
         MVC   Result,Data
         - - -
Data     DC    C'Data'
Result   DS    CL8
```

24.6.4.(1) What is in both operands after executing this MVC?

```
         MVC   Result2(8),Data2
         - - -
Result2  DS    C'ABCD'
Data2    DC    C'PQRSTUVW'
```

24.6.5.(2) In example 4 of Section 24.6.1, is there any reason (other than very poor style) not to write the last two instruction statements as

```
         STC   9,*+5
         MVC   LINE(0),0(8)    ?
```

24.6.6.(3) Consider these two examples of an MVCIN instruction with overlapping operands:

```
(1)      MVCIN X,Y+L'Y-1
         - - -
X        DS    CL7
         ORG   *-1
Y        DC    CL7'ABCDEFG'

(2)      MVCIN Q,P+L'P-1
         - - -
P        DC    CL5'12345'
         ORG   *-1
Q        DS    CL5
```

In each case, the target and source operands overlap by one byte. After executing the instructions, what data is at X and Q? Why is this one-byte overlap not a problem?

24.6.7.(2)+ Suppose three character strings are defined by the statements

```
A        DC    C'123456'
B        DC    C'PQRSTUVW'
C        DS    CL(L'A+L'B)
```

Write instructions to concatenate the strings at A and B into a single string at C.

24.6.8.(2)+ Suppose three character strings are defined by

```
D        DC    C'987654ABCDE'
E        DS    CL4
F        DS    CL(L'D-L'E)
```

Write instructions to split the string at D into two substrings at E and F.

24.6.9.(3)+ You are given a string of characters starting at **Str** whose length is stored at **N**, and are required to extract a substring of characters whose length is at **K**, starting at a character whose offset from **N** is stored at **P**. The extracted substring should be stored at **Sub**, and its length should be stored in the word at **L**. (N, K, P, and L are words in your program.)

In case the substring is not fully contained in **Str**, the values at K and L will differ; and if no valid substring can be extracted (for example, P exceeds N), store zero at L.

24.6.10.(2) A programmer needed to move a group of N records from **Source** to **Target**, where he assumed that the length of the record is defined by `L'Rec`. He wrote

```
         MVC   Target(N*L'Rec),Source
```

Under what circumstances will this work correctly, or not?

24.6.11.(2) What will happen in the instructions in Figure 200 on page 372 if the value of the Length Expression exceeds 256?

24.7. The NC, OC, and XC Instructions

The logical instructions NC, OC, and XC perform the AND, OR, and XOR operations described in Section 19 on two strings, byte by byte, leaving the result in the first operand string. The CC is set as in Table 93 on page 289. These examples illustrate the three instructions.

1. Clear the 120-byte area at **Line** to binary zeros.

   ```
   XC    Line(120),Line     Set 120 bytes to zero
   ```

   We could have used the same technique here as in example 1 of Section 24.6 (by moving a string of 120 zeroed bytes).

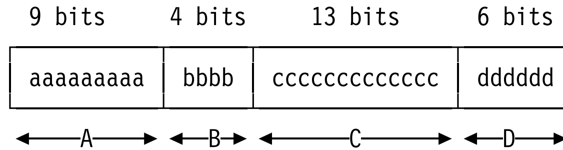
2. Branch to **Yes** if the fullword integer at **Lump** is zero.

```
            OC    Lump(4),Lump       OR 4 bytes to each other
            JZ    Yes                Branch if all bytes are zero
   or
            NC    Lump(4),Lump       AND 4 bytes to each other
            JZ    Yes                Branch if all bytes are zero
```

The first and second operands are identical, so only only the CC is set; no data is changed. This technique can sometimes be used when a register is not free.

Don't test a string of bytes for zero this way if the operand is memory-protected, because both instructions store into the first operand.

3. Suppose there are two words named **XX** and **ZZ** that each contain four positive integers, packed as illustrated in Figure 115 on page 249, shown here:



Replace the second integer in the word at **XX** by the corresponding value from the word at **ZZ**.

```
        MVC   Temp,ZZ            Move new value to temporary location
        NC    Temp,Mask          Eliminate all but second integer
        OC    XX,Mask            Set bits in 2d integer position to 1
        XC    XX,Mask            Now set them to zeros
        OC    XX,Temp            Insert new value into word at XX
        - - -
Temp    DS    XL4                Temporary workspace
Mask    DC    XL4'00780000'      Mask bits for 2nd integer position
```

Figure 204. Inserting bits in a word using logical SS-type instructions

4. Exchange the contents of the halfword integers at A and B. (Compare example 3 on page 373.)

```
            XC    B,A                XOR A to B
            XC    A,B                XOR B to A
            XC    B,A                XOR A to B
            - - -
   A        DS    H
   B        DS    H
```

This technique was used to exchange register contents in Exercise 19.5.1.*

Exercises

24.7.1.(2) Revise the instructions in Figure 204 to use two masks, two NC instructions, and one OC instruction.

24.7.2.(2) A student suggested the following code sequence as a solution to the problem of replacing data items embedded in a larger field:

* Which you solved correctly, of course.

```
         XC    Old,New            Make a mess of Old field
         NC    Old,Mask           Zero space for New item
         XC    Old,New            Now clean it all up
         - - -
Old      DC    C'DOWNWITH'
New      DC    C'PINKNUDITY'
Mask     DC    2X'FF',4X'0',2X'FF'
```

Verify that his method works, and discover the identity of the student.

24.7.3.(2) The bits in the byte at **BitData** are to be converted to a string of eight EBCDIC 0 and 1 characters starting at **BitChars**. A student suggested using these instructions:

```
         LHI   1,8                Count 8 bits in GR0
         IC    0,BitData          Get the source byte
Repeat   STC   0,BitChars-1(1)    Store a byte at BitChars
         SRL   0,1                Shift right by one bit
         JCT   1,Repeat           Iterate for all 8 bits
         NC    BitChars,=8X'1'    AND off all but low-order bit
         OC    BitChars,=8C'0'    OR makes EBCDIC 0 or 1 characters
         - - -
BitData  DC    B'10010001'        Sample source byte
BitChars DS    CL8                Converted characters
```

Does this work? What will be found at **BitChars**? Explain your answer.

24.8. The CLC Instruction

CLC compares the first operand to the second operand one byte at a time, until either an inequality is detected or the required number of bytes has been compared. As with CLI, each step of the comparison is between unsigned 8-bit logical integers, and the CC settings are as shown in Table 135 on page 355.

1. If the 120 bytes at Line contain blanks, branch to **AllBlank**.

```
         CLC   Line(120),=CL120' '     Compare to 120 blanks
         JE    AllBlank                Branch if equal
or
         CLC   =CL120' ',Line          Compare to 120 blanks
         JE    AllBlank                Branch if equal
```

   Because compare instructions modify neither operand, a literal can be used as the first operand; this second method uses the Length Attribute of the literal as the Length Expression.

2. Two non-negative word integers are stored at **SS** and **TT**. Branch to **TBig** if the number at **TT** is larger than the number at **SS**. (The restriction to non-negative integers means that a logical comparison gives the same result as an algebraic comparison.)

```
         CLC   TT(4),SS           Compare c(TT) to c(SS)
         JH    TBig               Branch to TBig if TT is greater
```

3. Two negative word integers are stored at **SS** and **TT**. Branch to **TBig** if the number at **TT** is algebraically larger than the number at **SS**. (Because both integers are algebraically negative, a logical comparison is the same as an algebraic comparison; see Exercise 24.8.1.)

```
         CLC   TT(4),SS           Compare logically, and...
         JH    TBig               Branch if c(TT) > c(SS)
```

4. A list of 100 names and occupations, each contained in a block of 60 bytes, is stored beginning at **List**. Branch to **Found** if any of the blocks matches the name and occupation in the block at **WhoIsIt**.

```
         LA    1,List            Initialize GR1 to A(first block)
         LA    2,100             Set GR2 count to number of blocks
Test     CLC   0(60,1),WhoIsIt   Compare blocks
         JE    Found             Branch if blocks are equal
         LA    1,60(,1)          Increment address by block length
         JCT   2,Test            Count down and branch
         J     NotFound          No matching block was found
```

Exercises

24.8.1.(2) Example 3 above claims that a logical comparison of two negative integers gives the same result as an algebraic comparison. Show that this is or is not true.

24.8.2.(3)+ Write instructions using CLC to correctly compare *arithmetically* two signed word integers in memory having arbitrary signs. For example, CLC should show that +10 > −10. (It can be done!)

24.8.3.(2) Suppose we wish to test the string of 220 bytes at **R** to see if they all contain zero. It is claimed that each of the following instructions will set the CC to zero if and only if the string contains all zero bytes. For which of these instructions is the claim true?

```
(1)      OC    R(220),R
(2)      NC    R(220),R
(3)      CLC   R+1(219),R
(4)      CLC   R(220),=220X'0'
```

24.8.4.(2)+ A programmer described the operation of the CLC instruction with the phrase “the shorter operand is padded with blanks”. Give two reasons why this is incorrect.

24.8.5.(2) Write instructions to test a string of 72 bytes at **Chars** and branch to **AllBlank** if every character is blank, without using a constant string of 72 blank characters. Use a CLC instruction.

24.8.6.(2)+ In Example 1 above, what would happen if you had written

```
         CLC   =120C' ',Line
```

instead?

24.8.7.(1)+ What does this instruction do? Is it at all useful? If so, why?

```
         CLC   1(7,4),0(4)
```

24.9. The TR (translate) Instruction

The Translate instruction replaces *each* byte in a string with *any* of another 256 possible values. Like MVC, the TR instruction moves bytes from the second operand location to the first operand location, but in a very different and possibly disorderly way. It actually performs a sort of “pseudo-indexing”:

1. An “argument” byte is obtained from the first operand address.
2. The value of that byte (as an eight-bit unsigned integer) is added internally to the second operand address, to access a “function byte” from the second operand.
3. The accessed function byte *replaces* the argument byte at the first operand address.[146]

---

[146] In mathematical terminology, the TR operation can be thought of as replacing the argument bytes x1, x2, ..., xn by the function bytes f(x1), f(x2), ..., f(xn). (Terminology aside, it's a simple operation.)

4. The first operand address is incremented by one, and the process repeats until all first operand bytes have been translated.
5. The Condition Code is unchanged.

For example, suppose the string of five argument bytes at **PP** contains X'0201040503', and the character string at **GG** contains the character constant C'ABCDEF'. If we execute the instruction

```
         TR    PP(5),GG
```

then the final contents of the five bytes at PP will be C'CBEFD'. The first argument byte taken from the first operand is X'02'; the function byte at GG+X'02' is C'C', and this replaces the first byte at **PP**. Similarly, the fifth and last argument byte at **PP** is X'03'; the function byte at GG+X'03' is C'D', which replaces the final byte in the string at **PP**.

Unlike the SS-type instructions we've seen thus far, the TR instruction can access bytes as far as 255 bytes away from the second operand address, whereas the other instructions accessed only those bytes within the area whose length is determined by the Length Specification Byte.

A sequence of RX-type instructions simulating the TR instruction helps clarify its operation. In Figure 205, the symbols **L**, **B1**, **D1**, **B2**, and **D2** have the values from the TR instruction being simulated. For this example, assume that **B1** and **B2** are not 1 or 2, because we will use those registers in the simulation.

```
*Emulate TR    D1(L,B1),D2(B2)    Translate L bytes
         LHI   0,L                Set counter in GR0 to number of bytes
         AHI   0,1                Get L; create program length N
         SR    1,1                Set first operand index to zero
         SR    2,2                GR2 Indexes table at 2nd operand
GetArg   IC    2,D1(1,B1)         Get argument byte from 1st operand
         IC    2,D2(2,B2)         Use as index to get function byte
         STC   2,D1(1,B1)         Store in string at 1st operand
         AHI   1,1                Increment first operand index by 1
         JCT   0,GetArg           Loop until N argument bytes done
```

Figure 205. Emulating the TR instruction

You can appreciate the power of TR if you consider the example in Figure 167 on page 332 and its variations. We wanted to replace all special characters with blanks. If we create an appropriate *translation* or *translate table*, the entire process can be done with *one* TR instruction, as in Figure 206.

```
         TR    Str(80),TRTable    Translate all specials to blanks
         - - -
TRTable  DC    (C'a')C' '         Anything less than C'a' is blanked
         DC    C'abcdefghi'       Letters are unchanged
         DC    7C' '              Non-printing characters are blanked
         DC    C'jklmnopqr'       Print letters as is
         DC    CL8' '             More non-printing characters
         DC    C'stuvwxyz'        Last of the lower-case letters
         DC    23C' '             Blank anything between 'z' and 'A'
         DC    C'ABCDEFGHI'       Letters are unchanged
         DC    7C' '              Non-printing characters are blanked
         DC    C'JKLMNOPQR'       Print letters as is
         DC    CL8' '             More non-printing characters
         DC    C'STUVWXYZ'        Last of the upper-case letters
         DC    6C' '              Blank anything between 'Z' and '0'
         DC    C'0123456789'      Digits print okay
         DC    6C' '              Tail-enders are blanked too
```

Figure 206. TR instruction to change special characters to blanks

As a second example of the TR instruction, suppose we will need to print the contents of the word at **HexWord** as eight hexadecimal digits, and we must place the eight EBCDIC characters representing the hex digits in a string starting at **Spred**.

```
         L     1,HexWord         Get fullword to be converted
         LA    2,Spred           Address of character being stored
         LA    3,8               Digit counter in GR3
Clear    SR    0,0               Clear GR0 for shifting
         SLDL  0,4               Shift a hex digit into GR0
         STC   0,0(,2)           Store in string at 'Spred'
         LA    2,1(,2)           Increment character address by 1
         JCT   3,Clear           Loop until 8 digits are stored
         TR    Spred,=C'0123456789ABCDEF'  Translate to EBCDIC
         - - -
Spred    DS    CL8               Converted result goes here
```

Figure 207. Translating hex digits to EBCDIC characters (1)

We can also index in the opposite direction, as in Figure 208.

```
         L     0,HexWord         Get fullword to be converted
         LA    2,8               Counter and index in GR2
Shift    SRDL  0,4               Shift a digit into GR1
         SRL   1,28              Position for storing
         STC   1,Spred-1(2)      Store in character string
         JCT   2,Shift           Decrease index and shift again
         TR    Spred,=C'0123456789ABCDEF'  Translate to EBCDIC
         - - -
Spred    DS    CL8               Converted result goes here
```

Figure 208. Translating hex digits to EBCDIC characters (2)

This result is sometimes called “spread hex”; the UNPK instruction (in Section 27) does this operation much more easily.

Exercises

24.9.1.(1) Assemble the translation table in Figure 206 on page 380 and verify that all non-blank characters are in positions corresponding to their EBCDIC encodings, and that the translate table is 256 bytes long.

24.9.2.(2)+ Suppose the bits within each byte of a string of bytes are to be rotated to the right by one bit position, so that B'10110001' becomes B'11011000'. Write a code sequence, including a TR instruction and the necessary translate table, to do the rotations.

24.9.3.(2) The translation table in Figure 206 on page 380 uses hand-counted values for the duplication factors on DC statements that generate blanks. Rewrite the table to use duplication factors calculated by the Assembler based on the hexadecimal representations of the characters.

24.9.4.(4) A certain program needed to place each of the 80 characters in the string at **InputRec** into an array of 80 words at **A1Format** in such a way that each successive word contains one character from the string in its leftmost byte, followed by three blank characters. (This format was used by some early Fortran compilers to read character data into a program.) Write a program segment (in Assembler Language, of course!) to do this using TR instructions.

24.9.5.(2)+ Write a short program segment which will use a TR instruction and an appropriate table to *interchange* the positions of the two hex digits in a byte. How long must the table be?

24.9.6.(2)+ Rewrite Exercise 22.5.1. to use a TR instruction and an appropriate translate table. Are there any limitations on the length of the list? Explain your conclusion.

24.9.7.(2)+ Rewrite Exercise 17.4.6 to use a Translate instruction and an appropriate translate table.

24.9.8.(4) Write a program segment to do the reverse of the action performed by your solution to Exercise 24.9.4: the high-order bytes of each of the fullwords in the array at **A1Format** should be collected into an 80-character string at **OutRec**.

24.9.9.(3) Assuming that only the valid EBCDIC characters shown in Table 13 on page 87 will appear in the string, write statements to generate a translation table that will set all other characters to blanks. Verify that your translate table is 256 bytes long.

24.9.10.(4) Write a sequence of instructions including TR that will cause the hex digits in a string of bytes at **Old** to be "shifted right" by *one* digit position at **New**. That is, if we start with X'123456' at **Old**, we should find X'012345' at **New**.

Then do the same for a left shift of one digit position; the result (starting with the same data) would then be X'234560'.

24.9.11.(2) Suppose you must translate a very long record to contain all upper-case letters. Assuming the record starts at **Record** and its length is found in the word at **Reclen**, write a translate table and instructions that will do the translation.

24.9.12.(2) Suppose you must convert the 8 hex digits of c(GR9) to 8 EBCDIC characters representing those digits, starting at **GR9Hex**. Will these instructions work? Explain why or why not, and describe the intended function of the instructions named **Q1**, **Q2**, and **Q3**.

```
         LHI   0,8
         LA    1,GR9Hex
Repeat   XR    8,8
         SLDL  8,4
Q1       AHI   8,240
Q2       CHI   8,250
         JL    Store
Q3       AHI   8,-57
Store    STC   8,0(,1)
         AHI   1,1
         JCT   0,Repeat
         - - -
GR9Hex   DS    CL8
```

24.9.13.(2)+ The string of characters at **Text** contains a mixture of lower-case and upper-case letters, and its length in the halfword at **TextLen** is less than 256. Write instructions including TR that will change the lower-case letters to their upper-case equivalents.

24.9.14.(2) Suppose the two bytes stored at **Zone** contain arbitrary bit patterns, represented as X'wxyz'. Write a code sequence with one or more TR instructions which will convert the given pair of bytes to the form X'Fxzy'. That is, interchange the two low-order digits, and replace the high-order digit by X'F', no matter what its original value might have been. (This is similar to the action performed by the UNPK instruction discussed in Section 27.)

24.9.15.(2)+ If you execute the following TR instruction, what will you find in the operand named **OddTable** when the instruction completes?

```
         TR    OddTable,OddTable  Identical first and second operands
         - - -
OddTable DC    X'01000302050405'
```

24.9.16.(3) A student suggested the following instructions as a way to convert a string of bytes at **InString** to pairs of EBCDIC characters at **OutStrng** representing the hex values of the source data. That is, if the first source byte contains X'9F', the first two output characters will be 9F.

```
         XR    0,0               Clear a work register
         XR    2,2               Input-byte index
         XR    3,3               Output string index
         LHI   4,L'InString      Number of bytes to convert
Convert  IC    0,InString(2)     Get a source byte
         SRDL  0,4               Shift right 4 bits
         STC   0,OutStrng(3)     Store leftmost hex digit
         SRL   1,28              Move rightmost hex digit to end
         STC   1,OutStrng+1(3)   Store rightmost hex digit
         AHI   2,1               Increment input index
         AHI   3,2               Increment output index
         JCT   4,Convert         Repeat for all input bytes
         TR    OutStrng,=C'0123456789ABCDEF'     Translate to EBCDIC
```

Does this work? What precautions should the student's program take?

24.9.17.(5)+ In Exercise 17.3.16 you reversed the bits in a 32-bit word, using shift instructions. Now, write a DC statement to create a translate table that will reverse the bits in each byte of a string.

24.9.18.(5) Using your solutions to Exercises 24.9.3 and 24.9.17, write a sequence of instructions using two TR instructions that will reverse both the bytes and the bits of the word at **DataWord** and store the result at **RevData**. For example, if c(DataWord)=X'12345678', the resulting c(RevWord) will be X'1E6A2C48'. (We'll see in Section 26 that there are easier ways to reverse bytes.)

24.9.19.(3) Suppose you define this translate table:

```
X        DC    256AL1(X'FF'-(*-X))
```

and the you execute this instruction:

```
         TR    X(256),X
```

What happens? If you repeat the instruction N times, will you ever get the original table at X? If so, how many times?

24.9.20.(3) Repeat Exercise 24.9.19 with this translate table:

```
X        DC    256AL1(*-X)
```

What happens? If you repeat the instruction N times, will you ever get the original table at X? If so, how many times?

24.10. The TRT and TRTR Instructions

Whereas TR *converts* a sequence of byte values into new values, TRT and TRTR are used to *search* a string of bytes for one or more specified values. These instructions are especially useful in scanning for punctuation, delimiters, and erroneous characters.

As we saw for MVC and MVCIN, the first operand of TRT refers to the *leftmost* byte of the first storage operand, while the first operand of TRTR refers to the *rightmost* byte of the first storage operand.

The operation of TRT and TRTR is identical to TR through steps 1 and 2 on page 379, and quite different thereafter.

3. The first operand is not modified; the accessed byte from the table addressed by the second operand (the function byte) does *not* replace the argument byte from the first operand string. Instead, the function byte is examined: if it is zero, we continue with step 4 of the description of TR, incrementing (or decrementing) the first operand address and decrementing the count.

   If the function byte is *not* zero,

   - It is placed in the rightmost byte of GR2 (the rest of the register is unchanged);

- The address of the argument byte which caused a nonzero function byte to be accessed is placed in GR1 or GG1, depending on the addressing mode:
  - in 24-bit mode, into the rightmost 24 bits of GR1, and the remaining bits of GR1 are unchanged
  - in 31-bit mode, into the rightmost 31 bits of GR1, and the leftmost bit of GR1 is set to zero
  - in 64-bit mode, into all the bits of GG1.
- The operation terminates, and the CC indicates the result of the operation, as shown in Table 147.

CC	Meaning
0	All accessed function bytes were zero.
1	A nonzero function byte was accessed before the last argument byte was reached.
2	The nonzero function byte accessed corresponds to the last argument byte.

Table 147. Condition Code settings for TRT and TRTR instructions

24.10.1. TRT

To illustrate the basic operation of TRT, suppose we must scan a string of characters to find the address of the first numeric character. First, we create a translate table with zero function bytes in all positions except for those corresponding to the EBCDIC representation of decimal digits, where the function bytes are nonzero.

```
NumChar  DC    (X'F0')X'00',10X'01',6X'00'
```

Then, suppose we test the following strings using TRT:

```
String1  DC    C'abc123def'       Decimal digit before end of string
String2  DC    C'*abcdef*'        No decimal digits
String3  DC    C'AB7'             Decimal digit in final position
```

Then, after executing the following instructions, the contents of GR1, GR2, and the CC are as shown:

Instruction	c(GR1)	c(GR2)	CC
TR String1,NumChar	A(String1+3)	X'xxxxxx01'	1
TR String2,NumChar	unchanged	unchanged	0
TR String3,NumChar	A(String3+2)	X'xxxxxx01'	2

The xxxxxx characters mean that those portions of GR2 are unchanged when the X'01' function byte is inserted.

As another example, suppose we must scan a string of 80 characters beginning at **Record** for the punctuation characters period, comma, and apostrophe. When one of them is found, a branch should be made to **Period**, **Comma**, or **Apost** respectively, with the address of that punctuation character in GR1. If none is found, branch to **NoPunct**. First, we will write an example using CLI instructions, but not TRT.

```
         LA    1,Record           Initialize character address
         LA    2,80               Number of characters to examine
TestPunc CLI   0(1),C'.'          Compare to period
         BE    Period             Branch if found
         CLI   0(1),C','          Compare to comma
         BE    Comma              Branch if found
         CLI   0(1),C''''         Compare to apostrophe
         BE    Apost              Branch if found
         AHI   1,1                Otherwise increment address by 1
         JCT   2,TestPunc         Count and loop
         B     NoPunct            Branch if none were found
```

Figure 209. Searching for punctuation characters using CLI

The TRT instruction does the same processing much more rapidly, but at the cost of memory space for the translate table.

```
         SR    2,2                Clear GR2, to be used as an index
         TRT   Record(80),PuncTbl Scan for punctuation
         JZ    NoPunct            Branch if none found
         B     *(2)               Function byte is index for branch
         J     Period             Period
         J     Comma              Comma
         J     Apost              Apostrophe
PuncTbl  DC    (C'.')X'00',X'04'       Function byte 4 for period
         DC    (C','-C'.'-1)X'00',X'08'             8 for comma
         DC    (C''''-C','-1)X'00',X'0C'            12 for apostrophe
         DC    (255-C'''')X'00'        Remainder of table
```

Figure 210. Searching for punctuation characters using TRT

The three nonzero function bytes are at positions in the table corresponding to the values of the EBCDIC representations of the characters being sought. The function values are multiples of four so they can be used to index the branch instruction `B *(2)`. If the conditional branch to **NoPunct** had been omitted, GR2 might contain zero and the program could have gone into an infinite loop at the B instruction.

This translate table was constructed by observing that we need not know the values of the EBCDIC representations of the period, comma, and apostrophe, only that their representations are in ascending order. This means that (for example) the number of characters between the period and the comma is the positive quantity `(C','-C'.'+1)`.

Suppose your program has received a string of decimal characters into a field named **InputNum**. Before using the data, it's a good practice to validate it. (Data validation helps avoid errors and program interruptions that may occur much later in your program.) Figure 211 shows one way to do this.

```
         TRT   InputNum,ValidDec  Test for all numeric data
         JZ    Valid              Branch to process valid data
         JNZ   ReEnter            Something invalid, ask for re-entry
         - - -
ValidDec DC    (C'0')X'01'        Values < X'F0' are invalid
         DC    10X'00',6X'01'     Values > X'F9' are invalid
InputNum DS    CL12               Numeric characters?
```

Figure 211. Using TRT to validate numeric characters

As another example of the TRT instruction, suppose we are required to scan the quoted character string starting at **Sentence** for the occurrence of an embedded character string containing either apostrophes (as in `"She said, 'Never!'"`), or quotation marks (as in `'I said, "I won't"'`). As these examples indicate, the *other* delimiter may appear freely inside the outer string.

If such an embedded string exists, we must store its starting address (excluding the preceding delimiter) at **StrAddr** and its length in bytes (again excluding the delimiters) at **StrLen**. (For the first string, the result would be the 10 characters She•said,•). If no string exists, branch to **None**, and if the apostrophe or quotation mark which would terminate the string is missing, branch to **Unfin**. Assume the length of the data string to be scanned is stored at **SentLen**, and is 256 or less. The program segment in Figure 212 scans first for the starting delimiter; when it is found, the proper table is chosen to search for the ending delimiter.

```
         LA    1,Sentence         Starting data address in GR1
         L     2,SentLen          Fetch length to scan, and...
         BCTR  2,0                Decrement by 1 for length byte,
         STC   2,TRT1+1           Store in TRT1 instruction.
         LA    3,0(2,1)           C(GR3) = A(last data byte)
         SR    2,2                Clear GR2 for function byte
         ST    2,StrLen           And set result length to 0
TRT1     TRT   0(*-*,1),T         Scan for first delimiter
         JZ    None               Exit if nothing useful found
         LA    4,1(,1)            Step over starting delimiter,
         ST    4,StrAddr          And store string start address.
LastCh   JC    2,Unfin            Exit if that's all there was
         LA    1,1(,3)            C(GR1) = A(last data byte)+1
         SR    3,4                (length-1) of rest of data
         STC   3,TRT2+1           Store in length byte of TRT2
         L     3,TAdd-4(2)        Address of correct table in GR3
         SR    2,2                Reset GR2 for function byte
TRT2     TRT   0(*-*,4),0(3)      Scan rest of data with new table
SetLen   S     1,StrAddr          Subtract start address of string
         ST    1,StrLen           And store result string length
         LTR   2,2                Test for closing delimiter found,
         JZ    Unfin              Branch if not found.
         - - -
StrLen   DS    F                  Length of final string
StrAddr  DS    A                  Address of final string
TAdd     DC    A(T2,T3)           Table addresses
T        DC    125X'0',X'040008',128X'0'    Initial TRT table
*        Function byte = 4 for apostrophe, 8 for quotation mark
T2       DC    125X'0',X'4',130X'0'    Stop on apostrophe
T3       DC    127X'0',X'4',128X'0'    Stop on quotation mark
```

Figure 212. Using TRT to scan for embedded quotations

Several items in Figure 212 deserve comment.

- The two STC instructions modify the TRT instructions at **TRT1** and **TRT2**. The Execute instruction in the next section shows a much better way to do this.
- The expression *-* in the instructions named **TRT1** and **TRT2** is the Location Counter value subtracted from itself, which is always zero. This notation is often used to indicate that the contents of the field will be provided by the program at execution time.
- By storing zero at **StrLen** before scanning the string (just preceding TRT1), we have taken care of the possibility that the initial delimiter may have been the last character in the data string; this condition is detected by the conditional branch instruction named **LastCh**.
- The function byte in the table named **T** is used by the Load instruction just preceding the second TRT as an index to load into GR3 the address at **TAdd** of the desired secondary table.
- By presetting GR1 to the address of the byte immediately following the data string, we can complete the scan with the second TRT as follows. If a closing delimiter exists, GR1 will eventually point to it, and the instruction named **SetLen** will calculate the number of bytes between the delimiters; GR2 will then contain the nonzero function byte X'04'. However, if no closing delimiter is found, GR1 and GR2 are *unchanged*, and we can still compute a useful string length before exiting to **Unfin**.

As a final example using TRT to scan variable-length data, suppose a string of characters at **Names** contains names separated by commas and terminated by a period. We will construct at **List** a table of fullword addresses of the first character of each name, followed by a word containing the number of characters in that name, which is known to be less than 256. When the table is complete, the number of names is stored in the word at **NbrNms**. To protect against omitted punctuation or other errors, we will branch to **LongName** if no comma or period is found within 256 characters of the start of a name. No tests are made for repeated names.

```
MaxNames Equ   50                  Assume at most 50 names found
         SR    3,3                 GR3 contains index for list
         SR    2,2                 Clear function-byte switch in GR2
         LA    1,Names             Initialize scan address
Scan     LR    4,1                 Save initial character address
         TRT   0(256,1),TRTB       Scan for period or comma
         JZ    LongName            Branch if no punctuation found
         ST    4,List(3)           Store address of name in list
         SR    1,4                 Compute name length
         ST    1,List+4(3)         Store length of name, too
         LA    3,8(,3)             Increment list index
         LA    1,1(4,1)            Move GR1 to start of next name
         JCT   2,Scan              Branch if comma was encountered
         SRL   3,3                 If period, compute and store ..
         ST    3,NbrNms            ...the number of names found
         - - -
TRTB     DC    (C'.')X'00',X'01'          Function = 1 for period
         DC    (C','-C'.'-1)X'00',X'02'  Function = 2 for comma
         DC    (255-C',')X'00'            Zero otherwise
         - - -
Names    DC    C'Brown,Green,Wonka,Ofstrand,Jones,Smedley,Doe,'
         DC    C'Apple,Doe,Smithwich,Softnard,Smith,Doelful,'
         DC    C'Lostkind,Jones,Lurp,VonHimmelsBergenSchneider,Doe.'
NbrNms   DS    F                   Number of names found
List     DS    (2*MaxNames)A       Table for addresses and counts
```

Figure 213. Using TRT to scan a string of names and build an occurrence list

The only unusual feature of Figure 213 is using the function byte as a branching switch: if a period is encountered, GR2 will contain +1, and the JCT instruction will not branch.

24.10.2. TRTR

The test in Figure 211 on page 385 can be done with TRTR and the same translate table:

```
         TRTR  InputNum+L'InputNum-1,ValidDec  Test for numeric data
         JZ    Valid               Branch to process valid data
         JNZ   ReEnter             Something invalid, ask for re-entry
        - - -
```

Figure 214. Using TRTR to validate numeric characters

Programs often must analyze character strings, finding “tokens” to be processed individually. But how do you know when there is no more data in the string, and the rest of the string is blanks? A common technique is to scan backwards from the end of the string, searching for the last non-blank character in the string. This is sometimes done with a CLI instruction:

```
         LA    1,String+L'String-1     Address of end of string
Check    CLI   0(1),C' '          Check for a blank
         JNE   Done               Exit loop if nonblank
         BCTR  1,0                Reduce address by 1 byte
         J     Check              And check again
Done     - - -                    GR1 points to last nonblank
```

Figure 215. Scanning a string backward using CLI

This scan can also be done (perhaps more quickly) using a TRTR instruction:

```
         TRTR  String+L'String-1,BlankTbl   Scan backward
         JZ    AllBlank           Problem: string is all blanks
         - - -                    GR1 points to last nonblank
BlankTbl DC    (C' ')X'1',X'0',(256-C' '-1)X'1'
```

Figure 216. Scanning a string backward using TRTR

While this may appear to use more memory than a CLI loop, translate tables like this are often used in many different places, so a single table can be referenced by many instructions.

Exercises

24.10.1.(1) Verify that the translate table in Figure 210 on page 385 generates exactly 256 bytes, and that the nonzero entries are at offsets corresponding to the EBCDIC representations of the punctuation characters.

24.10.2.(2) Show that the ORG instruction can be used to build the Translate and Test table of Figure 213 on page 387 as follows:

```
TRTB     DC    XL256'0'           Define table, length 256
         ORG   TRTB+C'.'
         DC    X'1'               Function byte for period
         ORG   TRTB+C','
         DC    X'2'               Function byte for comma
         ORG   TRTB+256           Reset LC to end of table
```

Use this technique to construct the table in Figure 210 on page 385. Why is this method superior to the one used in Figures 210 and 213? Can the first DC be replaced by DS?

24.10.3.(2) In Figure 212 on page 386, the length byte in the second TRT is calculated by the "SR 3,4" just preceding it. Is there any reason why the result of the subtraction cannot be negative? What would happen if it was?

24.10.4.(2) In Figure 212 on page 386, three distinct translate tables were used: one to scan for an initial apostrophe or quotation mark, and the other two to scan for the matching delimiter at the end of the quoted string. Rewrite the example to use a *single* translate table, which is suitably initialized for each use by instructions such as

```
         XC    T(256),T           Set entire table to zero
         MVI   T+C'''',4          Set to stop on apostrophe
```

24.10.5.(4)+ In Figure 212 on page 386, there are three translate tables. Show how these tables can be overlapped in a way that requires only about one-half as much space.

24.10.6.(2) Write instructions to scan the string of 120 characters at **CharData** and leave in GR1 the address of the first character that is neither alphabetic nor numeric.

24.10.7.(2) In Exercise 23.4.3 you scanned a character string at **Record** to locate the last non-blank character. Do the same exercise, but this time use MVCIN and TRT instructions instead of CLI.

24.10.8.(2) In Figure 212 on page 386, why is it necessary to reset GR2 to zero before executing the second TRT?

24.10.9.(2) The translate table in Figure 216 on page 388 is defined with a single DC statement. Verify that it generates the desired data.

24.10.10.(3) Modify the coding in Figure 213 on page 387 to store each name only once, and add a word to each name's entry in the list giving the number of occurrences of that word.

24.10.11.(2)+ Show the contents of GR2 and the Condition Code setting after executing the following instructions:

```
         SR    2,2
         TRT   XX,=XL5'20100'
         - - -
XX       DC    X'0004010203'
```

24.10.12.(3)+ Some experiments have shown that trailing blanks can be removed more efficiently than in Figures 215 and 216 by starting at the end of the string and comparing to a doubleword of blanks until a mismatch occurs, and then using a backward CLI scan. Write an instruction sequence to implement this technique to truncate the string starting at **String** having length L, and store the truncated length of the string in the halfword at **TruncLen**.

24.10.13.(2) Revise Exercises 24.7.3 and 24.10.7 to use a TRTR instruction.

24.10.14.(2)+ In Figure 215 on page 388, what will happen if the content of **String** is all blanks?

24.11. The Execute Instructions

While the Execute instructions are not SS-type, they are often used with SS-type instructions to help process character data.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
44	EX	RX	Execute	C60	EXRL	RIL	Execute Relative Long

Table 148. Execute instructions

The two Execute instructions in Table 148 are unusual, because they specify the execution of *another* instruction at a different address! We will use some concepts of the basic instruction cycle described in Section 4 and illustrated in Figure 13 on page 50.

These instructions are executed using these steps:

1. The Effective Address is computed, and the R_1 digit of the Execute instruction is saved.
2. The instruction at the Effective Address, the *target* (or *subject*) instruction, is placed into the Instruction Register (IR), replacing the EX or EXRL. The Instruction Address in the PSW is unchanged, and still contains the address of the instruction following the Execute.
3. If the new instruction in the IR is another execute instruction, a program interruption occurs, and the Interruption Code in the old PSW is set to 3. (There is a good reason for this interruption, as we'll see shortly.)
4. If the R_1 digit of the Execute instruction was zero, proceed to step 5. Otherwise, the rightmost byte of general register R_1 is ORed into the second byte of the IR. Both GR R_1 and the target instruction in memory remain unchanged.
5. The (possibly modified) target instruction in the IR is now decoded and executed as though it was the original instruction fetched from memory.

If the target instruction in the IR does not change the IA in the PSW (it is not a successful branch instruction), execution continues with the instruction following the Execute. If the target instruction does change the IA in the PSW (it is a successful branch), execution will continue with the instruction at the branch address. The CC is changed only if the target instruction sets the CC.

24.11.1. Execute Instruction Without Target-Instruction Modification

To illustrate uses of EX and EXRL, we first consider examples where the R_1 digit is zero, so that no ORing occurs in the IR.

1. Store at **CCC** the quantity 2*C(A)-C(B), where A and B are the names of words in memory.

```
        SR    1,1              Clear index to zero
        LA    2,4              Increment = 4, instruction length
        LA    3,12             Comparand = 12
Execute EX    0,Inst(1)        Execute an instruction
        JXLE  1,2,Execute      Increment by 4 and loop
        - - -
Inst    L     0,A              Load GR0 from A (4-byte instruction)
        AR    0,0              Double c(GR0) (2-byte instruction)
        NOPR  0                2 bytes spacing
        S     0,B              Subtract c(B) (4-byte instruction)
        ST    0,CCC            Store result (4-byte instruction)
```

Figure 217. Executing a list of instructions

   This program segment does four simple instructions the hard way, and merely illustrates a way to execute instructions which are "out-of-line", and not directly in the normal stream of program execution. The list of instructions at **Inst** could be executed independently of the first five instructions by branching to **Inst**, giving the same result much more rapidly.

2. Suppose we wish to add the three word integers stored beginning at **Q**. Depending on the number of overflows: if no overflows occur, multiply the result by 10; if one overflow occurs, do nothing; and if two overflows occur, set the result to 1.

```
        SR    1,1              Clear overflow counter in GR1
        L     0,Q              Get first integer
        A     0,Q+4            Add second integer
        JNO   NoOfloA          Branch if no overflow
        AHI   1,4              Indicate one overflow
NoOfloA A     0,Q+8            Add third integer
        JNO   NoOfloB          Branch if no overflow
        AHI   1,4              Indicate another overflow
NoOfloB EX    0,FixIt(1)       Execute correct operation
        ST    0,Result         And store result
        - - -
FixIt   MH    0,=H'10'         Multiply by 10
        NOP   0                Do nothing
        LA    0,1              Set result to +1
```

Figure 218. Executing a list of instructions

3. Suppose we must place in GR6 the address of some byte in memory, and that the desired address is known only to be the Effective Address of some other RX-type instruction. To make matters more complicated, suppose also that the addressing calculation needed by the RX instruction could make use of any registers but R14 and R15; that is, the base and index digits can be anything from 0 to 13. We assume that GR15 is currently being used for a base register, and that GR14 contains the address of the RX instruction in question.

   We will construct a LA instruction in a work area with the same index, base, and displacement fields as the RX instruction, and then execute that LA instruction.

```
         MVC   MakeLA(4),0(14)     Move original RX inst'n to work area
         NI    MakeLA+1,X'0F'      Clear old R1 digit position
         OI    MakeLA+1,X'60'      Set new R1 digit to 6
         MVI   MakeLA,X'41'        Set 'LA' opcode into instruction
*                                  Contents of MakeLA is now 416xbddd
         EXRL  0,MakeLA            Execute the constructed 'LA'
         - - -                     GR6 now has the desired address
MKLA     DS    2H                  4 bytes on halfword boundary
```

Figure 219. Constructing an executed instruction

This instruction sequence changes no registers other than GR6, even though R0 could have been used in the instruction sequence without affecting the operation of the EX, because GR0 or GG0 could not have been used by the LA as a base or index register. This illustrates a technique you can use when all other register contents must remain unchanged.

24.11.2. Execute Instruction with Target-Instruction Modification

The Execute instructions are most useful when the R_1 digit is not zero, implying modification of the target instruction in the IR.

1. Suppose we wish to move to **Line** a message whose address and length are in GR8 and GR9 respectively, as in example 4 on page 373.

```
         BCTR  9,0                 Decrease Length Expression by 1
         EX    9,Move              Execute the MVC instruction
         - - -
Move     MVC   Line(*-*),0(8)      Executed instruction, length = 0
```

   The Length Specification Byte in GR9 is ORed into the proper position in the (target) MVC instruction in the IR. In the assembled MVC instruction, the length byte was preset to zero by a zero explicit Length Expression. A major advantage of this method is that the instruction in storage is unmodified, an important consideration in writing re-enterable code.

   This is a very typical use of an Execute instruction.

2. Suppose we must branch to **Yes** if the rightmost byte of GR3 contains B'00011111'.

```
         EX    3,CLI               Execute the comparison
         BE    Yes                 Branch if equality is found
         - - -
CLI      CLI   ChkBits,0           Executed instruction
ChkBits  DC    B'00011111'         Comparison quantity
```

   The same problem could be solved without using EX, but extra storage accesses would be required:

```
         STC   3,Temp              Store the byte to be tested
         CLI   Temp,B'00011111'    Compare to desired pattern
         JE    Yes                 Branch if equal
         - - -
Temp     DS    X                   Byte from GR3 to be tested
```

3. Store at **RegTotal** the sum of the contents of registers GR0 through GR10.

```
         LA    12,10               Count in GR12
Loop     EX    12,Adder            Execute add instruction, sum in GR0
         JCT   12,Loop             Decrease counter and register digit
         ST    0,RegTotal          Store sum at RegTotal
         - - -
Adder    AR    0,0                 R2 digit modified by EX
```

   The R_2 digit of the AR instruction is modified in the IR to contain values from 10 to 1. It is rare to use Execute instructions to modify register specification or mask digits of executed instructions.

4. The fullword at **Mask** contains an integer whose value lies between 0 and 15, to be used as the mask digit of a BC instruction branching to **CondMet**.

```
         L     1,Mask              Get mask value
         SLL   1,4                 Position correctly as M1
         EX    1,BCInst            Execute the BC
NotMet   - - -                     Fall through if condition not met
         - - -
BCInst   BC    0,CondMet           BC with mask of 0
```

To complete execution of the EX instruction, the mask digit in the rightmost byte of GR1 is ORed into the BC instruction in the IR. The branch condition is now determined in the usual way; if it is met, the branch address of **CondMet** will be placed into the IA in the PSW. The execution of a successful branch instruction causes control to be "taken away" from the EX instruction.

5. Branch to **OddVal** if the rightmost bit of GR9 is a 1-bit (that is, the number in GR9 is odd).

```
         EX    9,TMInst            Execute a TM instruction
         JNZ   OddVal              Branch if a 1-bit
         - - -
TMInst   TM    OneBit,0            Test all 8 bits of next byte
OneBit   DC    B'00000001'         Only rightmost bit = 1
```

In the IR, the rightmost byte of GR9 becomes the mask byte (the immediate operand, I_2) of the TM instruction. This mask then tests whatever bits of the byte at **OneBit** correspond to 1-bits in the mask. If the rightmost bit of GR9 is a 1-bit, the tested bits will not be all zero, and the branch to **OddReg** will occur. (There are *much* easier ways to do this test!)

6. As a final (and more practical) example, suppose GR5 contains an integer specifying the number of bytes to be moved from a string beginning at **AA** to an area whose address is contained in GR7. The number of bytes may be greater than 256.

```
         LTR   5,5                 Test number of bytes to be moved
         BNP   Finis               Exit if not greater than zero
         LA    1,AA                GR1 contains 'from' address
Test     CHI   5,256               See if byte count exceeds 256
         JL    Last                If not, do last part of move
         MVC   0(256,7),0(1)       Move 256 bytes
         AHI   1,256               Increment 'from' address
         AHI   7,256               Increment 'to' address
         AHI   5,-256              Decrease byte count by 256
         JNZ   Test                If not zero, go try again
         J     Finis               If count is zero, all done
LMVC     MVC   0(0,7),0(1)         Move last part of byte string
Last     BCTR  5,0                 Decrease byte count by 1 for ex
         EX    5,LMVC              Move last part of string
Finis    - - -                     Rest of program goes here
```

Figure 220. Moving a string of bytes of unknown length

In Section 25, we will see how the MVCL and MVCLE instructions can handle such "long" moves more simply.

Using an EX instruction to supply the length byte for an SS-type instruction is its most common application.

24.11.3. Comments on the Execute Instructions (*)

1. The reason that an Execute instruction may not be the target of an Execute instruction (as stated in step 3 on page 389) is that the CPU could remain in a Fetch-Decode loop (comprising steps 1 through 4 of the Execute-instruction description) if the Execute instruction tried to execute itself, or if a chain of Execute instructions was circular. That is, consider what could happen if the instruction

```
         EX    0,*                 What are we doing, and why???
```

is allowed. This loop is very awkward for a CPU to stop, and is avoided simply by not allowing Executes of Execute instructions.[147]

2. When possible, place the target of an Execute instruction close to the EX.[148]
3. Be *very* careful when executing instructions with 12-bit opcodes, such as AHI. The low-order digit of the R_1 register should be zero, to avoid changing the opcode of the target.
4. Instructions that form Effective Addresses relative to an executed instruction use the address of the target, not the address of the Execute instruction itself. For example, suppose GR9 contains a branch mask value in the next-to-rightmost hex digit, and we execute a Branch Relative instruction:

```
         EX    9,GoToXYZ          Execute the relative branch
         - - -
GoToXYZ  BRC   *-*,XYZ            Branch if CC matches the mask bits
```

   The CPU determines the Effective Address of the BRC instruction relative to *its* address, not the address of the EX.

5. The Execute instruction was sometimes described as a special *branch* instruction! It is said that EX causes an unconditional branch to the target instruction, followed by an unconditional branch back to the instruction following the EX, unless the target instruction is itself a successful branch.

   This incorrectly describes the contents of the Instruction Address, which remains at the address of the instruction following the EX, and obscures the modification of the second byte of the target instruction. This is sometimes described by saying “the instruction is modified, but remains unchanged in memory”.

   Both descriptions are misleading.

While this discussion of the IR may not be exactly what's done in System z processors, it does describe the effect of the instruction, and gives a better feel for the way the CPU executes its instructions. You need not believe in the “magic” of an instruction being simultaneously modified and remaining unmodified.

24.11.4. Modifiable Parts of Instructions

The highlighted parts of the operands in the instructions listed in Table 149 on page 394 indicate the modifiable portions of typical instruction types as targets of the Execute instructions.

---

[147] This is a fact of computing life that was learned the hard way: on some early processors, the only “fix” was to turn off power and restart the machine. I knew a graveyard-shift computer operator on an older machine who discovered how to create an Execute loop. Then, he would file a “Computer Trouble Report” for the engineers, and go home early.

[148] Some programmers use an “idiom” like this:

```
LA    1,N                Number of bytes to move
BCTR  1,0                Make it a machine length
MVC   A(*-*),B           Move only 1 byte from B to A
EX    1,*-6              Move all N bytes from B to A
```

This is not generally recommended, because the CPU must process the MVC instruction twice.

Type	Operands	Modifiable
RR	$\boldsymbol{R_1},\boldsymbol{R_2}$	R_1,R_2
RX, RXY	$\boldsymbol{R_1},D_2(\boldsymbol{X_2},B_2)$	R_1,X_2
RS, RSY	$\boldsymbol{R_1},\boldsymbol{R_3},D_2(B_2)$ $\boldsymbol{R_1},\boldsymbol{M_3},D_2(B_2)$ $\boldsymbol{R_1},D_2(B_2)$	R_1,R_3 R_1,M_3 R_1
SI, SIY	$D_1(B_1),\boldsymbol{I_2}$	I_2
SS	$D_1(\boldsymbol{L},B_1),D_2(B_2)$ $D_1(\boldsymbol{L_1},B_1),D_2(\boldsymbol{L_2},B_2)$ $D_1(\boldsymbol{L_1},B_1),D_2(B_2),\boldsymbol{I_3}$	L L_1,L_2 L_1,I_3
SSF	$\boldsymbol{R_3},D_1(B_1),D_2(B_2)$	R_3
RI	$\boldsymbol{R_1},I_2$ $\boldsymbol{M_1},I_2$	R_1 M_1

Table 149. Modifiable portions of typical EX target instructions

Because the second byte of some instructions contains part of the operation code, there is usually little reason to execute those instructions with a nonzero R_1 digit.

Exercises

24.11.1.(2) What is the relationship between the USING statements in effect when an EX instruction is assembled, and those in effect when the target instruction is assembled?

24.11.2.(2)+ A programmer believed that EX “branches to the target instruction, and then branches back to the instruction following the EX if the target instruction was not a successful branch”. Consider the following code sequence:

```
         EX    0,BASR1
Here     - - -
         - - -
BASR1    BASR  1,0
There    - - -
```

What would he claim to be in GR1 after the EX is executed? What *will* be in GR1?

24.11.3.(2)+ Suppose control passes to the following sequence of instructions:

```
         LA    1,BStar
         EX    0,BASR1            Execute the BASR instruction
         LR    0,0                Do nothing in particular
BStar    B     *                  Wait here for answer
BASR1    BASR  1,0                Do something, maybe
         AR    0,0                Also do nothing in particular
         B     BStar              Branch to waste some cycles
```

When control arrives at **BStar**, the address of some instruction should be in GR1. What is it? What will be the value of the Instruction Length Code immediately *after* the EX instruction has completed execution?

24.11.4.(2)+ Rewrite Exercise 17.2.9 to use an EX instruction and eight TM instructions to test the proper bit of the selected byte.

24.11.5.(3) We must move a number of bytes from a string whose starting address is contained in GR1, to a string whose starting address is contained in GR2. The number of bytes to be moved (which can be greater than 256) is in GR3. Write a code sequence to perform the move.

24.11.6.(3)+ Suppose you must scan a string of length L (where L $\le$ 200) bytes starting at **DCData** that may contain paired ampersands and apostrophes (as in a C-type character con-

stant). Write instructions to scan the string and move it to **DCGen** with each paired occurrence replaced by a single occurrence. Store the length of the resulting string at **DCGenL**.

24.11.7.(2) In Exercise 24.9.13 on page 382, the string at **Text** was assumed to be shorter than 256 characters. Repeat the exercise, now assuming that the string's length may be up to 14000 characters.

24.11.8.(3)+ A string of M characters at **Data** is to be moved to an N-byte area named **DataPad** and extended or “padded” with blanks if N > M. Assume that both M and N are ≤256, and have been defined in EQU statements. If N < M, move only N bytes. (Don't use MVCL or MVCLE!)

24.11.9.(3) In Exercise 24.8.4 on page 379, you explained why CLC does not pad the shorter operand with blanks. Write an instruction sequence that simulates the operation of a “CLC” instruction that *does* pad the shorter operand with blanks. Your instructions must set the Condition Code correctly.

24.11.10.(3) Parentheses are used in many programming languages to enclose expressions, denote groupings, and so forth. These parentheses must be *balanced*: that is, they must “match up” so that (1) each left parenthesis has a matching right parenthesis that follows it somewhere, (2) the leftmost parenthesis must be a left parenthesis, and (3) it must be matched by the rightmost parenthesis. More formally, if L(n) and R(n) are the number of left and right parentheses encountered after scanning n characters, and if there are N characters in the string, then a balanced string must have L(n)≥R(n) for 0<n<N, and L(N)=R(N).

Using appropriate TRT and EX instructions, write a program segment which will test a string of characters for balanced parentheses. Assume initially that GR7 contains the address of the string, and its length in bytes is a 32-bit binary integer in GR8. Branch to **Balanced** and **Unbalncd** for successful and unsuccessful scans, respectively.

24.11.11.(2)+ Modify the illustration of the fetch-decode-execute cycle in Figure 16 on page 55 to show how the Execute instruction and its target instruction are fetched and decoded. Indicate explicitly where the test for an Execute exception is made.

24.11.12.(2) Suppose the bits within the byte stored at **Rotator** are to be rotated to the right N bit positions, where N is defined by the three low-order bits in GR1. Write a code sequence, including the necessary translate table or tables, to do the shift.

Can you devise a table which will accomplish the shift by executing only a single TR instruction?

24.11.13.(2)+ In example 5 on page 392 in Section 24.11.2, we want to test for the presence of a 1-bit in GR9. What will happen if the branch instruction is JO instead of JNZ?

24.11.14.(2) How can the code sequence in example 5 of Section 24.11.2 be modified to test if the contents of some register is a multiple of a given power N of 2? What are the limitations on this technique?

24.11.15.(2)+ A programmer used an EX instruction to load the constant 137 into a general register whose number was determined at execution time. He knew that the number of the target register would be in the rightmost 4 bits of GR1, and wrote

```
         EX   1,LHIOp             Load the constant into a GPR
         - - -
LHIOp    LHI  0,137               Executed: load into the target GPR
```

This won't do what he wants. Explain why not, and show what he should have written.

24.11.16.(2) Write a code fragment using an Execute instruction that will convert the byte at **Byte** to 8 EBCDIC characters starting at **Char** that represent the value of its 8 bits.

24.11.17.(2) Modify the coding in Figure 212 on page 386 to use EX instructions where appropriate.

24.11.18.(3)+ Suppose your CPU has no MVCIN instruction, and you want to move the string of L characters starting at **Source** to the string of L characters starting at **Target** in reverse order. Assuming that 0 < L < 256 is a number in GR0, create an appropriate translate table and instructions that will move the characters as required.

24.11.19.(2)+ If c(GR1) = X'FEDCBA98', and you then execute this instruction:

```
         EX    1,Sub
         - - -
Sub      SR    5,2
```

what SR instruction will the CPU actually execute?

24.11.20.(2) How can an EX instruction choose one of multiple possible target instructions in a single execution?

24.12. Summary

Remember:

The length you code in an SS-type assembler instruction statement (**N**) specifies how many bytes are involved (unless you code zero, in which case one byte always participates in the operation). The length you specify as the R_1 operand of an EX instruction (**L**) is one less than the number of bytes involved.

For most instructions, operand overlap is not a problem.

- If neither operand is changed (for example, by CLC and TRT), operand overlap doesn't matter.
- Most instructions operate as though the bytes of the source operand are fetched one at a time, and the result byte is stored at the target operand before the next source byte is fetched.[149]
- For other instructions, operand overlap can lead to unpredictable results.

We'll note special cases as they arise.

Table 150 summarizes Table 142 on page 367 and Table 143 on page 368 about explicit and implied length specification in single-length SS-type instructions:

Explicit Length	Implied Length
$S_1(N),S_2$	S_1,S_2
$D_1(N,B_1),S_2$	$D_1(,B_1),S_2$
$S_1(N),D_2(B_2)$	$S_1,D_2(B_2)$
$D_1(N,B_1),D_2(B_2)$	$D_1(,B_1),D_2(B_2)$

Table 150. Operands of single-length SS-type instructions

The instructions discussed in this section are shown in Table 151 on page 397.

[149] Modern processors may fetch, process, and store groups of several bytes, but the result still appears to be byte-at-a-time operation.

Function	Instruction	Data is Processed	CC Set?
Move	MVC MVCIN	Left to right Right to left	No
Move	MVCOS	Left to right	Yes
AND	NC	Left to right	Yes
OR	OC	Left to right	Yes
XOR	OC	Left to right	Yes
Compare	CLC	Left to right	Yes
Translate	TR	Left to right	No
Translate and Test	TRT	Left to right	Yes
Translate and Test Reverse	TRTR	Right to left	Yes
Execute	EX EXRL	—	Depends on target

Table 151. Basic instructions for data in storage

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
CLC	D5
EX	44
EXRL	C60
MVC	D2

Mnemonic	Opcode
MVCIN	E8
MVCOS	C80
NC	D4
OC	D6

Mnemonic	Opcode
TR	DC
TRT	DD
XC	D7

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
44	EX
C60	EXRL
C80	MVCOS
D2	MVC

Opcode	Mnemonic
D4	NC
D5	CLC
D6	OC
D7	XC

Opcode	Mnemonic
DC	TR
DD	TRT
E8	MVCIN

Terms and Definitions

IR
: Instruction Register; an internal register holding a target instruction so its second byte may be modified by an Execute instruction prior to decoding.

Length Expression
: A value (**N** or **LE**) coded implicitly or explicitly in a machine instruction statement for an SS-type instruction, from which the Assembler derives the *Length Specification Byte* **L**.

Length Specification Byte
: The second byte (**L**) of an SS-type instruction, one less than the length of its operand or operands.

target instruction
An instruction addressed by an Execute instruction.

Programming Problems

Problem 24.1.(2) Using the definitions in Exercise 24.11.10, write a program which will read character strings from records and test for balanced parentheses. Print each string and a message which indicates whether or not it is balanced. Assume that the first blank character ends the character string.

Problem 24.2.(3) A "perfect shuffle" of a deck of 52 playing cards interleaves each card of the top 26 cards with each card of the bottom 26, in exactly the same way for each shuffle.[150] Thus, after a single shuffle the order of the cards is 1, 27, 2, 28, ... 25, 51, 26, 52.

It is claimed that after a small number (less than 10) of perfect shuffles, the original order of the cards is restored. Test this claim by writing a program using a TR instruction to perfectly shuffle the numbers from 1 to 52. The results may be displayed in hexadecimal.

Just for fun, try your program for different (even!) numbers of "cards" to see how many shuffles are needed to recover the original order.

Problem 24.3.(3) Write a program to read 80-character records that should contain only EBDCIC decimal digits or blanks. If any invalid character is found, display the record and the column number where the invalid character was found.

Problem 24.4.(4) In storing data containing large numbers of characters, it is often useful to find some way to "compress" the data. For example, if the data consists of 80-byte records that rarely contain 80 nonblank characters, space might be saved if we discard the trailing blanks (following the last nonblank character on the record, and place a *control byte* at the beginning that gives the length of the remaining string. Thus, a record containing only an asterisk in column 1 would be stored as the two bytes `X'015C'`, a saving of 78 bytes.

Many such compression schemes exist, and they can be simple or elaborate depending on the needs of a particular situation. One packing method applicable to strings containing many repeated characters is the following:[151]

1. Copy the first character of the record in its exact form.
2. Replace each subsequent character by the binary value (or *difference*) that when added to the preceding character's *value* (ignoring carries) will produce the desired character.
3. When the difference is zero, indicating a redundant (repeated) character, that zero difference is used as a *flag*, and the *next* character contains a count of the number of remaining repetitions.
4. Preceding each string of compressed text is a *record length byte* containing the number of bytes in the string, *including* itself.
5. A record length byte containing zero indicates the end of the compressed text for that record.

These examples may help:

```
Input String        Compressed Line Record

AAAAAAAAAA          04 C1 00 08
AAAAABBBBB          07 C1 00 03 01 00 03
BBAA                06 C2 00 00 FF 00 (no extra 00!)
ABCDDDDDEFFGH       0D C1 01 01 01 00 03 01 01 00 00 01 01
```

[150] This is also known as an "out-shuffle".

[151] Due to J. E. Hunter, IBM Technical Disclosure Bulletin, Volume 15, Number 6, November 1972.

(This compression scheme will require *more* space than the original text if the longest string of repeated characters is of length 2, as in the third example.)

Write a program that will read 80-byte records and produce a block of compressed text records in memory. For example, if there were only a single compressed text record (a single “line” of text), as in the fourth example above, then the block in memory would contain 14 bytes:

```
0D C1 01 01 01 00 03 01 01 00 00 01 01 00
```

Then, “de-compress” the block of text, and print it. If you can, write your compressed text onto records and give them to someone else to expand. Compare the expanded results to the original records.

Problem 24.5.(4)+ A common requirement in scanning character strings is that some character positions must match a “pattern” exactly, while other positions may match any character. For example, suppose a pattern is defined by C'AB%CD', meaning that when scanning a test string, AB must match the first two characters of the test string, the % means that any character in the third position of the test string is acceptable, and CD must match the fourth and fifth characters of the test string. For example, this pattern would match the test string C'AB?CD'. but not the test string C'ABCDEF'.

Write a program that will accept a pattern string (perhaps on a record with initial characters 'Pattern') followed by records containing test strings. Print the pattern, the test string, and an indication of whether the pattern matches the test string or not.

Problem 24.6.(5)+ This problem is an extension of Problem 24.5. In addition to a pattern character like % that will match an arbitrary character in a test string, it is often useful to match some characters while some number of others need not be matched. For example, if a pattern uses the character * to mean “match any number of characters, including none”, then the pattern C'A*B' would match test strings like C'AB' and C'A123B'. Similarly, a pattern like C'A*' would match test strings like C'A' and C'ABCDEFG'.

As in Problem 24.5, write a program that will accept various patterns followed by test strings, and print the pattern, the test string, and an indication of whether the pattern matches the test string or not.

Problem 24.7.(5)+ Combine the two pattern types of Problems 24.5 and 24.6, so that a pattern might look like C'A%B*C', which would match test strings like C'A.BC' and C'AJBRSTUVWC'.

Problem 24.8.(2) A programmer claimed that he can convert the hex digits of a word stored at **Word** to eight EBCDIC characters stored at **HexWord** representing the hex digits with these instructions:

```
         L     2,Word
         L     4,=4X'0F'
         LR    3,2
         SRL   3,4
         NR    3,4
         NR    4,2
         STM   3,4,Temp
         TR    Temp,=C'0123456789ABCDEF'
         MVC   HexWord,=X'0004010502060307'
         TR    HexWord,Temp
         - - -
Temp     DS    D
HexWord  DS    CL8
Word     DC    X'1F2E3D4C'        (For example)
```

Write a program to test his claim with a variety of values at **Word**.

Problem 24.9.(2) Write a program to simulate the execution of the TR instruction.

Problem 24.10.(3) Write a program to simulate the execution of the TRT instruction, without considering final Condition Code settings

Problem 24.11.(4) Write a program to simulate the execution of the TRT instruction, with correct settings of the Condition Code when execution completes.

Problem 24.12.(3)+ Write a program to display on three lines the 80-byte records of any short program you wrote: on the first line, the original record (with double-spacing carriage control; and on the second and third lines, each byte of the original line is shown in “vertical hex”, where the first hex digit of the EBCDIC character is shown on the upper line, and the second hex digit on the lower line. For example, the characters `This is a TEST` (or `X'E38889A24088A2408140E3A5E2E3'`) would be arranged on 3 output records like this:

```
0This is a TEST
 E88A48A484EAEE
 38920820103523
```

The key to the solution is the translate table.

This technique can be very useful for displaying the contents of records (like object modules) that contain a mixture of EBCDIC and binary values.

Problem 24.13.(3) Write a program to read the records shown below. The first record is a title line, the next is blank, and the remaining records contain the name of a student and four exam grades in columns 31-40, 41-50, 51-60, and 61-70.

Write a program to read the data and produce a report with the average grade for each student in columns 71-80, and after the last student's grades and average, skip a line and print the average grade for each exam. For example, the output might be formatted like this:

```
Name                       Exam 1    Exam 2    Exam 3    Exam 4   Average

Student 1                      nn        nn        nn        nn        nn
 - - -
Student n                      nn        nn        nn        nn        nn

Exam Averages                  nn        nn        nn        nn        nn
```

This is some sample data:

```
Name                       Exam 1    Exam 2    Exam 3    Exam 4     Final

Doaks, Joe                     79        83        88        91        93
Queue, Susie                   44        91        67        97        89
Shakes IV, Pete                97        89        80       100        73
Burley, Hurley                 61        71        85        88        97
Throckmorton, Chauncey         90        90        88        74        92
Doaks, Jonathan                79        83        87        95        47
```

Problem 24.14.(3) Write a program to read the records shown below, that contain the text of Lincoln's “Gettysburg Address” as a string of characters in fixed-format 80-byte records. You will need a work area of about 2000 bytes.

Count and print the number of words.

These are the records for you to read:

```
Four score and seven years ago our fathers brought forth on this continent a ne
w nation, conceived in liberty, and dedicated to the proposition that all men a
re created equal. Now we are engaged in a great civil war, testing whether that
 nation, or any nation, so conceived and so dedicated, can long endure. We are
met on a great battle-field of that war. We have come to dedicate a portion of
that field, as a final resting place for those who here gave their lives that t
hat nation might live. It is altogether fitting and proper that we should do th
is.

But, in a larger sense, we can not dedicate, we can not consecrate, we can not
hallow this ground. The brave men, living and dead, who struggled here, have co
nsecrated it, far above our poor power to add or detract. The world will little
 note, nor long remember what we say here, but it can never forget what they di
d here. It is for us the living, rather, to be dedicated here to the unfinished
 work which they who fought here have thus far so nobly advanced. It is rather
for us to be here dedicated to the great task remaining before us -- that from
these honored dead we take increased devotion to that cause for which they gave
 the last full measure of devotion -- that we here highly resolve that these de
ad shall not have died in vain -- that this nation, under God, shall have a new
 birth of freedom -- and that government of the people, by the people, for the
people, shall not perish from the earth.
```

Problem 24.15.(4) Write a program to read the same records as in Problem 24.14. Now, create a table of distinct words, ignoring differences between lower and upper case forms of the same word. Sort the words into alphabetical order, and print the words and the number of occurrences of each.

Problem 24.16. Write a program to read the same records in Problem 24.14.

Your program should then create a readable version of the text on lines of 60 characters, with words correctly joined where they were split across the original records. No characters may go past the end of the 60-character output line. For example, if two input records contain something like this:

```
◄─────────────────────────────── 80 characters ───────────────────────────────►
word1 word2 word3 word4 word5 word6 word7 word8 word9 wordiness wordA wordB word
C wordD wordE ...
```

then your formatted line would contain something like this:

```
◄────────────────────── 60 characters ─────────────────────►
word1 word2 word3 word4 word5 word6 word7 word8 word9
wordiness wordA wordB wordC wordD wordE ...
```

If you encounter a completely blank line in the input, leave a blank line in the formatted output.

You might enjoy making the width of the output line depend only on a symbol defined by an EQU statement; try values such as 60, 80, and 100.

Problem 24.17.(4) Write a program to read the data in Problem 24.14. Create an output area of the same size as your input area. Now, you will encrypt the records from the input to the output areas. Create a "key" by defining a suitably random binary fullword from a 9-digit decimal value. Then, encrypt the input message as follows:

1. XOR your key with the first word of the input message, and store the result in the output buffer.
2. XOR that result with the second word of the input message, and store the result in the second word of the output buffer.
3. Continue in this way until the entire message has been encrypted.

Then, create a third buffer of the same length as the input buffer. Using your key, *decrypt* the message in the output into this third buffer, using the same technique. Then, compare your decrypted result with the original. (They should be identical!)

Problem 24.18.(3) Write a program to read records with blank-terminated strings of octal (base 8) digits. Assuming the octal digits represent a right-adjusted binary number, convert the digits to binary represented as a string of hexadecimal digits. Then, display the original octal digit string followed by the converted hex string.

Remember that the rightmost octal digit contains the three low-order bits of the binary number, so that octal 76543 (`O'76543'`? What's its value in decimal?) is the same as `X'7D63'`.

25. Character Data and Extended Instructions

```
 2222222222    55555555555
222222222222  555555555555
22        22  55
          22  55
          22  555555555
         22    5555555555
       22              555
     22                 55
   22                   55
 22                    555
222222222222  55555555555
222222222222  555555555
```

All the instructions discussed thus far complete their task before the next instruction is processed. In this section, we'll look at some instructions that may take much longer to complete. This means that if the CPU is interrupted by a high-priority request, something must be done about the current instruction. The CPU handles this in one of two ways:

- Method A: Save enough information in the general registers about the intermediate state of the instruction, reset the Instruction Address back to the address of the interrupted instruction, and then process the interrupt. When execution of your program resumes, the interrupted instruction continues from its intermediate state as though it had not been interrupted.
- Method B: Process a portion of the operands, update registers appropriately, and set the Condition Code to 3 to indicate that the operation was only partially completed. Any pending interruptions can then occur. The Instruction Address is not reset, so the following instruction should test for CC=3 and branch back to the interrupted instruction so it can complete the operation.

We'll see each “method” in the instructions in this section.

The processed portion of the operands can vary greatly from instruction to instruction, and for repeated executions of the same instruction.

25.1. Move Long and Compare Logical Long

These instructions also use a special “padding” character, or an “end” (or “stop”, “test”, “search”, “special”, “terminating”) character.[152] All the instructions in Table 152 on page 404 use a *padding* character.

[152] The appropriate name for the “end” character depends on how it's used; some names are more descriptive than others for a given instruction.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
0E	MVCL	RR	Move Long	A8	MVCLE	RS	Move Long Extended
0F	CLCL	RR	Compare Logical Long	A9	CLCLE	RS	Compare Logical Long Extended

Table 152. Basic character-handling instructions using padding characters

We begin by examining the Move Long (MVCL) and Compare Logical Long (CLCL) instructions in a general way. Both are RR-type instructions with the usual format, and they both use *four* general registers! The instruction formats are

```
MVCL  R1,R2
CLCL  R1,R2
```

where *both* R_1 and R_2 designate an even-odd pair of registers. (A specification exception occurs if either R_1 or R_2 is odd.) The even-numbered registers contain the operand addresses, and the next-higher odd-numbered registers contain the operand lengths; each length is treated as a 24-bit unsigned number. The high-order byte of R_2+1 contains the *padding byte*, as sketched in Figure 221. (Register lengths and addressing modes are ignored for a moment.)

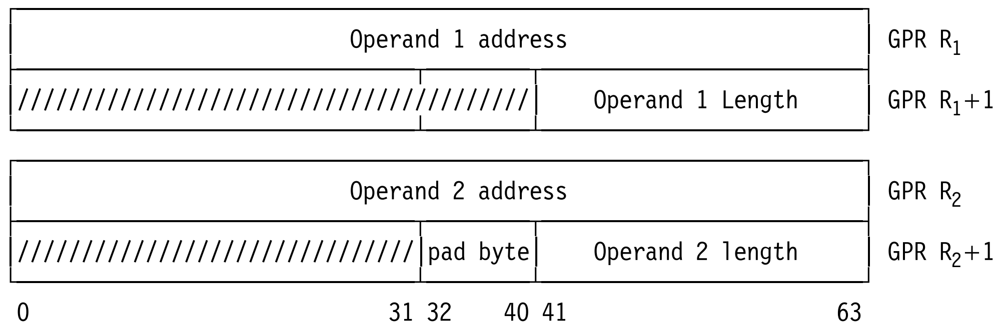


Figure 221. Register use by CLCL and MVCL

MVCL and CLCL simplify moving and comparing long strings of bytes, which would otherwise require lengthy loops.

These instructions are unlike MVC and CLC in several respects.

- *Two* lengths are specified; the operands may have different lengths, and the instructions depend on both lengths.
- Much longer strings of bytes may be compared or moved in a single instruction. Instead of a limit of 256 bytes, MVCL and CLCL can specify up to $2^{24}-1$ bytes.[153]
- All four registers may be changed by the instructions. The addresses in the even-numbered registers depend on the addressing mode.
- The lengths are true lengths rather than “machine lengths” (the true length minus 1). Only the 24 low-order bits of R_1+1 and R_2+1 (containing the operand lengths) are updated, and the remaining bits are unchanged.
- The high-order byte of R_2+1 holds a “pad byte” used to extend certain operands, if necessary.
- The MVCL instruction sets the Condition Code, and no data movement takes place if there is any possibility of “destructive overlap” of the operands.
- Either R_1 or R_2 may be zero, so that GR0 may contain an operand address!
- Both instructions are interruptible: if an interrupt occurs before the operation is complete, “Method A” is used: the registers are updated appropriately, and the Instruction Address in the PSW is “backed up” by 2 bytes from the address of the following instruction to the

[153] At the time MVCL and CLCL were implemented, The operand length of $2^{24}-1$ bytes seemed sufficient for many years. But memory sizes grew rapidly, so MVCLE and CLCLE were added to handle longer compare and move operations. We'll see them in Section 25.2.

address of the CLCL or MVCL instruction. When control is returned to the interrupted program, execution of the instruction resumes with the remnants of the operands.

25.1.1. MVCL

In the absence of special conditions, MVCL operates by moving bytes from the second (source) operand field to the first (target) operand field. As noted in our discussion of implied lengths in Section 24.3, the number of bytes moved is controlled by the *first* (receiving) operand length. Thus, if the first operand length is zero, no bytes are moved.

Unlike MVC, MVCL tests for the possibility of *destructive overlap*, which occurs when any part of the first operand field is used for a source after data has been moved into it. If destructive overlap could occur, the CPU sets the Condition Code to 3, and moves no data.

Execution of MVCL proceeds (conceptually) as follows:

1. Bytes are moved one by one from the second to the first operand field; the counts are decremented by 1 and the addresses are incremented by 1 for each byte moved.
2. If both operand counts reach zero at the same time, the CC is set to 0.
3. If the first operand count reaches zero before the second operand count, the CC is set to 1. That is, the target length in R_1+1 less than the source length in R_2+1.
4. If the second operand count reaches zero first, the pad character is used as a source byte until the first operand count reaches zero; the CC is then set to 2. That is, the target length in R_1+1 greater than the source length in R_2+1.
5. On termination, the operand 1 length is zero, and the operand 1 address has been updated by the corresponding length. The operand 2 address has been incremented by the number of bytes moved from the second operand field whether or not padding has occurred, and the second operand count has been decreased by the same amount.
6. On termination (even for destructive overlap),
   - in 24-bit addressing mode, the leftmost byte of GR R_1 and GR R_2 are zeroed, and the high-order half of GG R_1 and GG R_2 are unchanged;
   - in 31-bit addressing mode, the leftmost bit of GR R_1 and GR R_2 is zeroed, and the high-order half of GG R_1 and GG R_2 are unchanged;
   - in 64-bit addressing mode, both GG R_1 and GG R_2 are updated.

MVCL sets the Condition Code as shown in Table 153.

CC	Meaning
0	Operand 1 length = Operand 2 length
1	Operand 1 length < Operand 2 length; part of operand 2 not moved
2	Operand 1 length > Operand 2 length; operand 1 was padded
3	Destructive Overlap, no data movement

Table 153. CC settings after MVCL

Figure 222 on page 406 may help you to to visualize the operation of MVCL, assuming there is no destructive overlap. The figure uses these notations:

A1 Address of a first-operand byte, c(R_1)
c(A1) The first-operand byte at address A1
L1 Remaining length of the first operand, c(R_1+1)
A2 Address of a second-operand byte, c(R_2)
c(A2) The second-operand byte at address A2
L2 Remaining length of the second operand, c(R_2+1)
Pad Padding byte

Reader Note

In the following figure sketching the flow of the MVCL instruction, there are many places where subscripts would be more appropriate, but the formatter used for this text cannot then properly align other parts of the diagram. This comment also applies to Figures 225, 229, 232, 235, 236, and 239 in this Section 25.

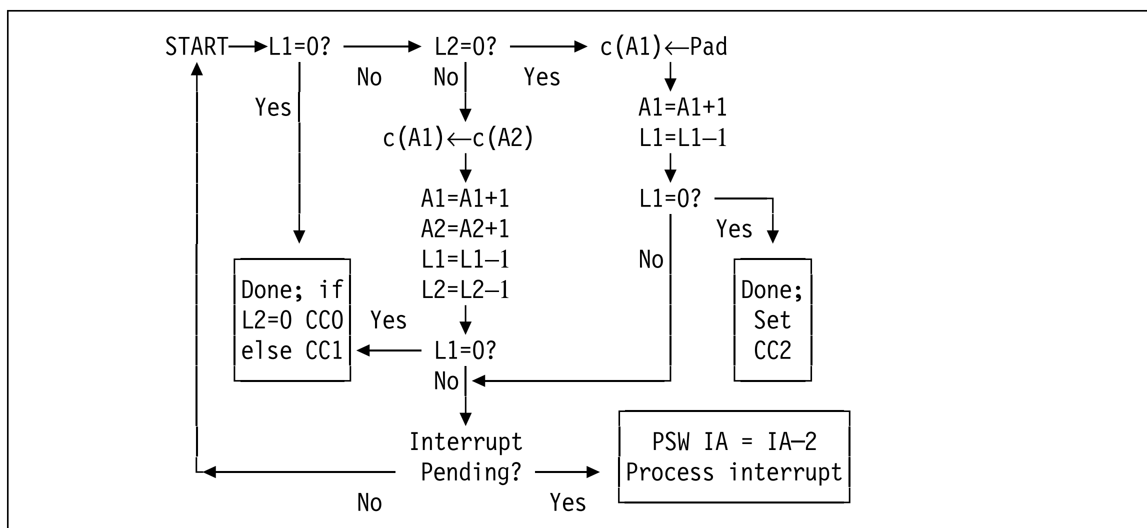


Figure 222. Conceptual execution of the MVCL instruction

To illustrate some uses of MVCL, suppose we want to set the area at **Line** to blanks (as in example 1 of Section 24.6).

```
LineLen  Equu  120               Number of blanks to move
         SR    1,1               Operand 2 length = 0
         ICM   1,8,=C' '         Pad character is blank
         LA    2,Line            Set address of first operand in GR2
         L     3,=A(LineLen)     And first operand length in GR3
         MVCL  2,0               Move pad characters to 'Line'
```

Figure 223. Using MVCL to set a field to blanks

Because the second operand length in GR1 is zero, we need not initialize GR0 with an address.

This method is a lot more work than the example in Section 24.6, because we must set up four registers and a pad byte. However, MVCL is superior to MVC when the number of bytes to be moved grows large: by omitting the ICM (which inserts the padding character into R1), we could just as easily have set the area to zero, as in example 1 of Section 24.7. MVCL is often used this way to zero large blocks of memory without having to use XC instructions, and to initialize areas without using MVC instructions with overlapping operands.

Suppose GR8 and GR9 contain the address and length of a message which is to be moved to the 120-byte area at **PrintMsg**. We will pad the message with blanks if it fits, and branch to **WontFit** if all of the message won't fit in the 120-byte area.

```
         LA    2,PrintMsg        First operand address
         LA    3,L'PrintMsg      First operand length
         ICM   9,8,=C' '         Set padding character to blank
         MVCL  2,8               Move the string, pad if necessary
         JL    WontFit           Branch if something left over (CC1)
         JO    NotMoved          Error, destructive overlap (CC3)
```

Figure 224. Moving a message with padding and length checking

No Execute instruction (or STC, to store a length byte into an MVC) was needed to supply a Length Specification Byte for the move.

25.1.2. CLCL

The two operand byte strings are compared byte by byte as unsigned binary numbers (just as for CLC), starting at the low-addressed end and proceeding toward higher addresses. The comparison stops when an inequality is detected, or when the end of the *longer* operand is reached (not the shorter!). Unlike MVCL, where only the first operand might be padded, CLCL can pad *either* operand! The CC is set in the usual way to indicate the result of the comparison. If both operand lengths are zero, or if R_1 and R_2 designate the same register, the CPU simply sets the CC to zero, indicating equality.

The comparison can be considered as proceeding in the following way:

1. Bytes are compared one by one; the operand addresses are incremented by 1 and the operand lengths are decremented by 1 for each step.
2. If an inequality is detected before either length becomes zero, the CC is set, registers R_1 and R_2 contain the addresses of the unequal bytes, and the counts in the rightmost 24 bits of the respective odd-numbered registers contain one more than the number of bytes that remain to be compared. (That is, the addresses have been incremented by the number of equal bytes, and the lengths have been decremented by the same amount.) The CC setting indicates the larger or smaller operand.
3. If one of the lengths becomes zero, the comparison continues with the padding character being compared to bytes from the longer operand. For the shorter operand, the even register contains the address of the first byte past the end of the operand string, and the odd register contains zero.
4. If an inequality is detected between the padding character and a byte from the longer operand, the address and count for that operand are set as in step 2 (the address and count for the shorter operand were set in step 3).
5. If no inequality is detected before the longer count becomes zero, the even-numbered register points to the first byte past the end of the longer operand string.
6. The register contents on termination are the same as shown for MVCL in step 6.

If the two operands are completely equal (including the padding character, if needed), both counts will be zero, and the corresponding addresses will have been incremented by the original count values.

CLCL sets the Condition Code as shown in Table 154.

CC	Meaning
0	Operand 1 = Operand 2, or both lengths 0
1	First Operand low
2	First Operand high

Table 154. CC settings after CLCL

Figure 225 on page 408 may help clarify this description. The figure uses notations similar to those preceding Figure 222 on page 406:

A1 Address of a first-operand byte, c(R_1)
c(A1) The first-operand byte at address A1
L1 Remaining length of the first operand, c(R_1+1)
A2 Address of a second-operand byte, c(R_2)
c(A2) The second-operand byte at address A2
L2 Remaining length of the second operand, c(R_2+1)
Pad Padding byte
x:y x is compared to y

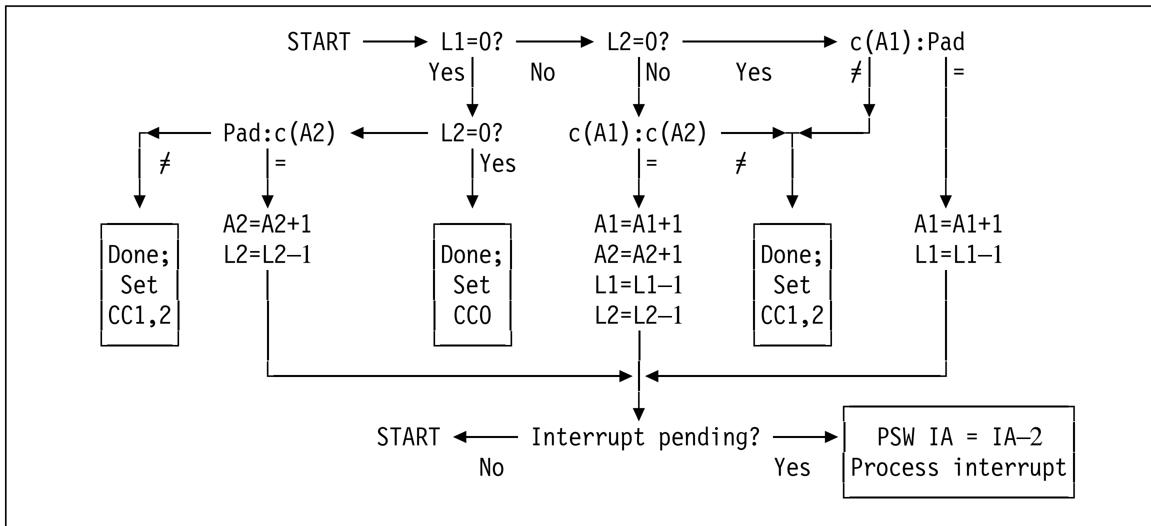


Figure 225. Conceptual execution of the CLCL instruction

The greater power of CLCL compared to CLC is seen in the changes to the four registers at the end of the operation. Not only do you know exactly how many bytes were compared, but the precise position of the inequality is known, which is impossible with CLC unless the bytes are compared one at a time.[154] By testing the lengths for zero, you can tell whether an inequality occurred between bytes in memory, or between the padding character and a byte in memory.

To illustrate CLCL, suppose we want to see if all 120 bytes at **Line** contain blanks, and branch to **AllBlank** if so.

```
LA    2,Line           First operand address in GR2
LA    3,120            First operand length in GR3
SR    1,1              Second operand length = 0
ICM   1,8,=C' '        Pad character is blank
CLCL  2,0              Compare first operand to blank
JE    AllBlank         Branch if all blanks at 'Line'
- - -                  GR2 points to first nonblank char
```

Figure 226. Using CLCL to test for blanks

Because the length of the second operand in GR1 is zero, no address is needed in GR0.

Suppose we have read two records into memory, and want to determine if they are equal; let the addresses and lengths of the records be stored in fullwords at **Addr1**, **Len1**, **Addr2**, and **Len2** respectively.

- If the records are unequal up to the length of the shorter record, we will branch to **UnEqual**.
- We branch to **Equal** if their lengths *and* contents are identical.
- We branch to **Equal1** or **Equal2** respectively if the operands are equal up to the shorter length, but operand 1 or operand 2 is longer.
- Neither operand may be padded.

[154] For long strings of bytes, this could be painfully slow.

```
         L     2,Addr1           Set first operand address
         L     3,Len1            And length of first record
         L     6,Addr2           Set second operand length
         L     7,Len2            And length of second operand
         LA    1,Equal           Assume lengths are equal
         CLR   3,7               Compare lengths
         JE    Compare           Go compare if equal lengths
         JH    Op1Long           Branch if operand 1 longer
         LA    1,Equal2          Operand 2 longer, set equality exit
         LR    7,3               2nd operand length = shorter value
         J     Compare           And compare
Op1Long  LA    1,Equal1          Operand 1 longer, set equality exit
         LR    3,7               1st operand length = shorter value
Compare  CLCL  2,6               Compare operands with equal lengths
         JNE   UnEqual           Branch if inequality detected
         BR    1                 Branch to desired equality routine
```

Figure 227. Comparing two records without padding

The preliminary effort in this example ensures that GR3 and GR7 will both contain the shorter operand length when the CLCL is executed. This illustrates precautions we must take if we don't want the shorter operand to be extended with a padding character.

Exercises

25.1.1.(1) Why does using GR0 for an operand address not violate the rules given in Section 10, where GR0 can't be used to generate an Effective Address?

25.1.2.(2)+ What do you think will happen if MVCL is the target of an EX instruction, and an interruption occurs before the MVCL operation is completed?

25.1.3.(1) Using MVCL, is there any possibility of destructive overlap if the length of the second operand is 1?

25.1.4.(1) Suppose the operand 1 length of an MVCL instruction is zero. What will be the CC setting?

25.1.5.(2)+ A 3500-byte area at **Field** contains a value in its first byte that is to be propagated through the rest of the area (all 3500 bytes are to contain that value). Write a code sequence using MVCL to perform the task.

25.1.6.(3)+ It is claimed that destructive overlap will *not* occur with MVCL if

- operand 1 address ≤ operand 2 address, or
- operand 1 address > operand 2 address + MINLEN − 1

where MINLEN is the smaller of the two operand lengths. Is this true? Why?

25.1.7.(2)+ Suppose we execute this instruction sequence:

```
LA    0,STR1          Address of first operand string
LA    1,L'STR1        Length of first operand string
LA    2,STR2          Address of second string
LA    3,L'STR2        Length of second string
ICM   3,8,PAD         Padding character in GR3
CLCL  0,2             Compare STR1 to STR2
```

For each of the following sets of definitions of the symbols **STR1**, **STR2**, and **PAD**, show what will be in GR0 through GR3 and the CC setting after the CLCL is executed. (Assume that the address of the symbol **STR1** is X'074212D0' in each case.)

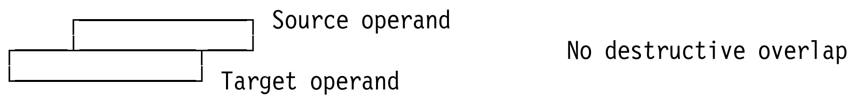
```
(1)      STR1  DC    F'1'
         STR2  DC    F'2'
         PAD   DC    X'0'

(2)      STR1  DC    CL120' '
         STR2  DC    110C' '
         PAD   EQU   STR2

(3)      STR1  DC    CL20'**'
         STR2  DC    C'**'
         PAD   DC    H'40'

(4)      STR1  DC    0XL5
         STR2  DC    C'ABCD'
         PAD   DC    X'FF'
```

25.1.8.(2) Sketching MVCL operands and their lengths sometimes helps you understand when destructive overlap may occur. For example, in this sketch,



no destructive overlap occurs because no target operand byte is used as a source operand byte if data is moved one byte at a time. Sketch other possibilities to determine when destructive overlap will and will not occur.

25.1.9.(3) As in Exercise 24.11.15, use one or more MVC instructions to move a string of bytes whose address is in GR1 to an area whose address is in GR2; the number of bytes in GR3 is greater than zero and less than `X'FFFFFF'`. Perform these additional tests:

1. If the strings overlap *destructively* branch to **Destroy** with no data having been moved.
2. If the data strings overlap, but no data is destroyed, perform the move and then branch to **Overlap**.

25.1.10.(3) What factors must be considered if you write instructions to emulate the behavior of CLCL?

25.1.11.(2) Rewrite the example in Figure 226 on page 408, reversing the operands: that is, make the first operand have zero length.

25.1.12.(1) Revise Exercise 24.8.5 to use a CLCL instruction.

25.1.13.(1) Can you do a "ripple" move with MVCL, as you can with MVC?

25.1.14.(1)+ Revise Figure 223 on page 406 to initialize to zero the 8192 bytes starting at **New**.

25.1.15.(4) Suppose you are using a CPU that does not support the MVCL instruction. Write instructions (not using MVCL!) to simulate MVCL, including correct Condition Code settings.

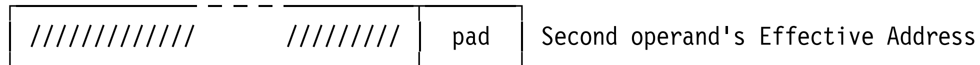
25.2. Move Long and Compare Logical Long Extended

These two instructions are only slightly more complicated than MVCL and CLCL. Both instructions use "Method B" to allow the CPU to process interruption conditions. Table 155 gives their format:

opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

Table 155. Format of MVCLE and CLCLE instructions

There are three operands: the operands in the even-numbered registers R_1 and R_3 are analogous to the R_1 and R_2 operands of MVCL and CLCL. The second operand is not used as an address; instead, the low-order 8 bits of its Effective Address are used as the padding byte.



Note that the second operand of the machine instruction is specified as the *third* operand of the assembler instruction statement.

For example, to specify a blank padding character, you could write

```
MVCLE 2,8,C' '(0)
CLCLE 4,14,X'40'(0)
```

Another difference is that the odd-numbered registers hold the 32-bit (or 64-bit) operand lengths, which can be from 0 to $2^{32}-1$ (or from 0 to $2^{64}-1$ for 64-bit addressing mode).

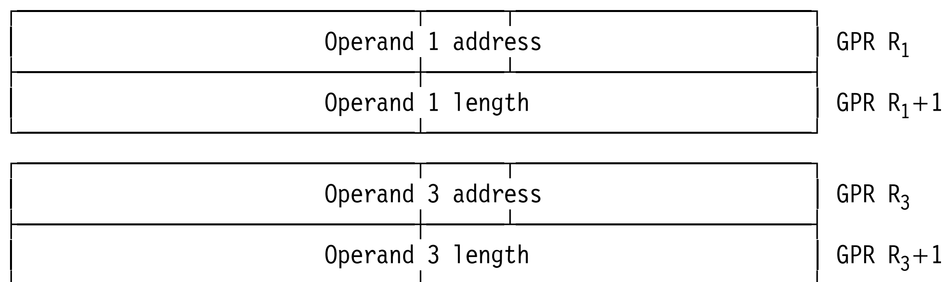


Figure 228. Register use by MVCLE and CLCLE

If these instructions are executed in 24- or 31-bit addressing modes, the rightmost 24 or 31 bits of the even-numbered registers contain the operand addresses. On termination, the high-order (non-address) bits of the R_1 and R_3 registers may or may not be set to zero.[155]

25.2.1. MVCLE

Unlike MVCL, no overlap test is done for MVCLE; the results of overlapping operands are unpredictable. Its execution is sketched in Figure 229 on page 412, using similar notations as in Figure 222 on page 406, except that the source operand address is A3 and the source operand length is L3.

[155] This is not just a whim on the part of the CPU; it's meant to give the CPU designers more freedom to decide how best to implement the instructions. (Maybe it's a whim on the part of the CPU *designers*?)

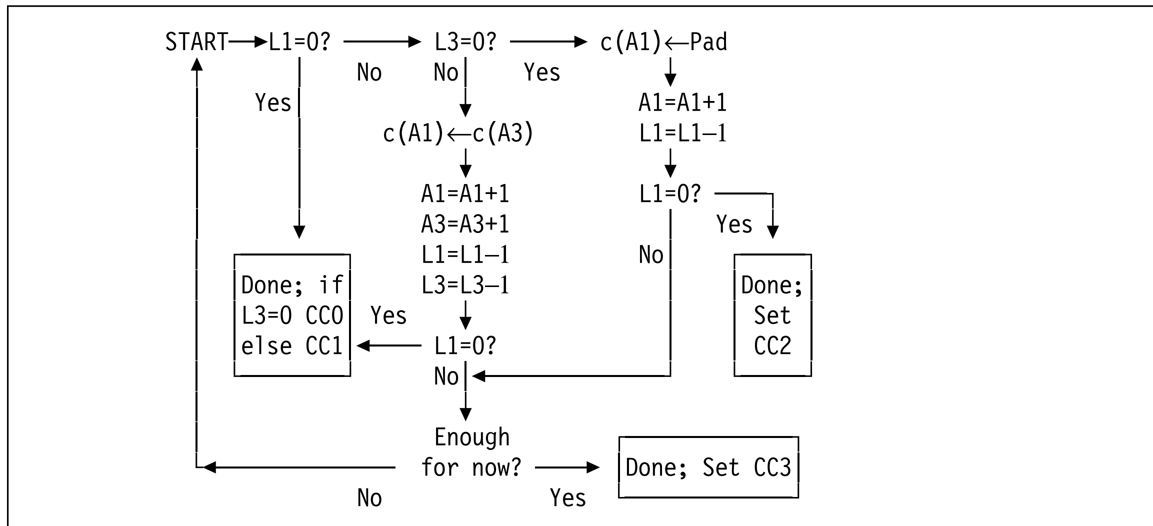


Figure 229. Conceptual execution of the MVCLE instruction

On termination, the register contents are:

1. If the first operand has been completed, the CC is set to 0 if the two operand lengths were equal. Both operand addresses have been incremented by that length, and both lengths are now zero.
2. If the first operand has been completed, the CC is set to 1 if the first operand is shorter than the third. Both operand addresses has been incremented by the first operand length, the first operand length is 0, and the third operand length has been decremented by the first operand's original length.
3. If the first operand has been completed, the CC is set to 2 if the first operand is longer than the third (meaning that the first operand was padded). Both operand lengths are zero, and both addresses have been updated by their original lengths.
4. If the CPU has moved enough bytes and wants to pause for any pending interruptions, the CC is set to 3. You would then branch back to the MVCLE instruction to resume the move. Both addresses and lengths have been updated for the bytes moved.
5. Whatever the reason for termination, the registers are updated to account for the amount of data that has been moved.
6. Some special padding byte values can be used to improve the performance of MVCLE; see the *z/Architecture Principles of Operation* for details.

To summarize, MVCLE sets the Condition Code as shown in Table 156.

CC	Meaning
0	All bytes moved, operand lengths are equal
1	All bytes moved, operand 1 shorter; part of operand 2 was not moved
2	All bytes moved, operand 1 longer; operand 1 was padded
3	Some bytes moved; end of operand 1 not reached

Table 156. CC settings after MVCLE

We can use MVCLE for the same task as in Figure 223 on page 406, where we again assume that GR8 and GR9 have been initialized already:

```
         LA    2,PrintMsg         First operand address
         LA    3,L'PrintMsg       First operand length
         LA    0,C' '             Set padding character to blank
Move     MVCLE 2,8,0              Move the string, pad if necessary
         JO    Move               Repeat if not finished
```

Figure 230. Using MVCLE to set a field to blanks

As another example, suppose we use MVCLE to initialize a large area of storage starting at **Work** to zeros:

```
         XR    0,0                Source address will be ignored,
         XR    1,1                ...because source length is zero
         LA    2,Work             Start of area to initialize
         L     3,WorkSize         Length of work area
Clear    MVCLE 2,0,X'00'          Initialize with X'00' padding
         JO    Clear              Repeat if necessary

BlockLen Equ   32000
NBlocks  Equ   20000
WorkSize DC    A(BlockLen*NBlocks)     Large area
```

Figure 231. Using MVCLE to initialize an area to zero

It's unlikely that you'd ever need to initialize such a large an area of storage; because the value at **WorkSize** is larger than 2^{24}, we can't use MVCL.

It may or may not be important to your application that MVCLE does not check for overlap.

25.2.2. CLCLE

CLCLE operates in much the same way as CLCL, as sketched in Figure 232, using the same notations as in Figure 225 on page 408, except that the source operand address is A3 and the source operand length is L3.

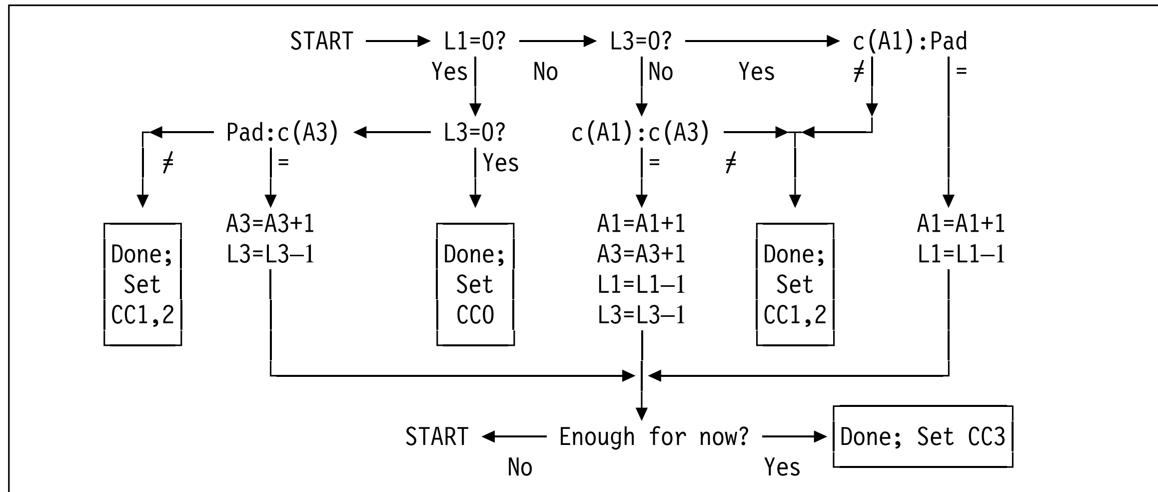


Figure 232. Conceptual execution of the CLCLE instruction

On termination, the registers are set as follows:

1. If an inequality is found, the CC is set as shown in Table 159 on page 415.
2. If the operands are equal (including the padding byte, if used), the addresses and lengths are updated to account for the number of bytes compared.

3. If the CPU has compared enough bytes without an inequality and wants to pause for any pending interruptions, the CC is set to 3. You would then branch back to the CLCLE instruction to resume comparing.

CLCLE sets the Condition Code as shown in Table 157.

CC	Meaning
0	All bytes compared, operands equal, or both zero length
1	First operand low
2	First operand high
3	Some bytes compared without finding an inequality

Table 157. CC settings after CLCLE

To illustrate, we'll rewrite Figure 221 on page 404 to use CLCLE:

```
        LA    2,Line            First operand address in GR2
        LA    3,120             First operand length in GR3
        XR    0,0               Second operand address is ignored,
        XR    1,1               ...because its length is zero
Compare CLCLE 2,0,C' '          Compare first operand to blanks
        JE    AllBlank          Branch if all blanks at 'Line'
        JO    Compare           Repeat if comparison is incomplete
        - - -                   GR2 points to first nonblank char
```

Figure 233. Using CLCLE to test for all blanks

The similarities of CLCL and CLCLE are close; only the operand lengths and the source of the padding byte are different. But, the differences between CLC and CLCL/CLCLE are more significant:

- CLC requires only one or two base registers to address the operands; CLCL/CLCLE both require up to four registers.
- CLC is limited to 256-byte operands; CLCL/CLCLE operands can be much longer.
- CLC simply indicates an inequality; CLCL/CLCLE also set R_1 and R_2 (or R_3) to the addresses of the unequal bytes.
- CLC does no padding; CLCL/CLCLE support a padding character.

Exercises

25.2.1.(2) What do you think will happen if the B_2 register of a CLCLE or MVCLE instruction is the same as R_1 or R_3 or R_1+1 or R_3+1?

25.2.2.(1)+ Can both operands of CLCL or CLCLE be padded?

25.2.3.(2) In 24-bit addressing mode, the maximum valid address is X'00FFFFFF', or $2^{24}-1$. However, MVCLE allows you to specify operand lengths up to X'FFFFFFFF', or $2^{32}-1$. What do you think will happen if you execute MVCLE with a length longer than X'FFFFFF'?

25.2.4.(1) In Figure 231 on page 413, what is the hex value of the word at **WorkSize**?

25.2.5.(2) Let NB2M be the number of bytes to be moved to **Tgt** from the second operand field at **Src** by MVCL. Make a table which gives the initial and final register contents, and the value of NB2M, for each of the possible resulting CC values when all bytes have been moved. Then, do the same for MVCLE.

25.3. Special "C-String" Instructions

The four instructions in Table 158 arose from the need to process character strings used by the C and C++ programming languages, where character strings are terminated by a zero byte (X'00') called a *null* byte.[156] The instructions have many general uses, whatever the origins of the data, and whether or not it contains a null terminating byte.

We will use a bold italic letter "n" to represent a null byte, as in "***n***". For example,

```
         DC    C'A C-string.',X'0'     Generates 'A C-string.n'
```

The length of a C-string does not include the terminating null character, so that the single byte X'00' represents a C-string of length zero (a "null string").

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B255	MVST	RRE	Move String	B25D	CLST	RRE	Compare Logical String
B25E	SRST	RRE	Search String	B2A5	TRE	RRE	Translate Extended

Table 158. Character-handling instructions for terminated strings

These instructions all have RRE format, as shown in Table 159:

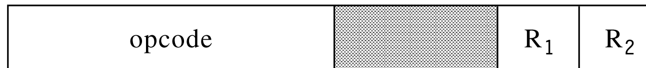


Table 159. Format of RRE-type instructions

Each instruction uses a special (end, test, or terminating) character in the rightmost byte of GR0. All but TRE require that the remaining bits of GR0 be zero.

The operation of the MVCL, MVCLE, CLCL, and CLCLE instructions is controlled by a length in a register; the four instructions in Table 158 are controlled by the presence of the special character in one or both operands. Only TRE uses both a length and a terminating character.

Exercises

25.3.1.(1)+ Write DC statements defining C-strings of length zero, one, and ten.

25.4. Search String Instruction

The SRST instruction is the simplest of these four instructions. It scans the second operand string addressed by register R_2, looking for a byte matching the specified "test" character in GR0. If a matching byte is found, the R_1 register is set to its address. Because the second operand string can be very long, the CPU uses "Method B" (described on page 403) to process part of the string before checking for interruptions.

For finding a single character, SRST is simpler and faster than a Translate and Test instruction like TRT, or a CLI loop. Unlike TRT, however, it searches for only a single character.

To use SRST, set the test character in GR0, set the R_2 register to the address of the leftmost byte of the string to be scanned, and the R_1 register to the address of the first byte *after* the end of the string. The CPU uses the address in the R_1 register to know when to stop the scan; otherwise, it could keep scanning bytes in memory until it found a match somewhere, or caused an unexpected interruption. This is summarized in Figure 234 on page 416.

[156] The earliest implementations of C were done on machines with instructions that could move bytes and simultaneously test their values, so very few instructions were needed to move null-terminated character strings.

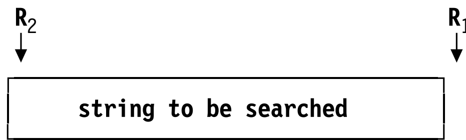


Figure 234. Registers bounding the SRST search string

Table 160 gives the Condition Code settings after SRST:

CC	Meaning
1	Test character found; R_1 points to it
2	Test character not found before the byte addressed by R_1
3	Partial search with no match; R_1 unchanged, R_2 points to next byte to process

Table 160. CC settings for SRST instruction

On completion, either or both of the R_1 and R_2 registers may be updated:

- If the CC is 1, the R_1 register is updated and the R_2 register is unchanged.
- If the CC is 2, both the R_1 and R_2 registers are unchanged.
- If the CC is 3, R_1 is unchanged and R_2 is updated to the address of the next byte to be tested. You can then branch back to the SRST instruction to continue the search.

When a register is updated, any high-order bits not used for addressing are set to zero. Figure 235 sketches the operation of the SRST instruction. The notation used in the figure is:

A1 Address of the first byte *after* the end of the string being searched, in R_1
A2 Address of a byte being checked during the search
Test Test character, taken from GR0

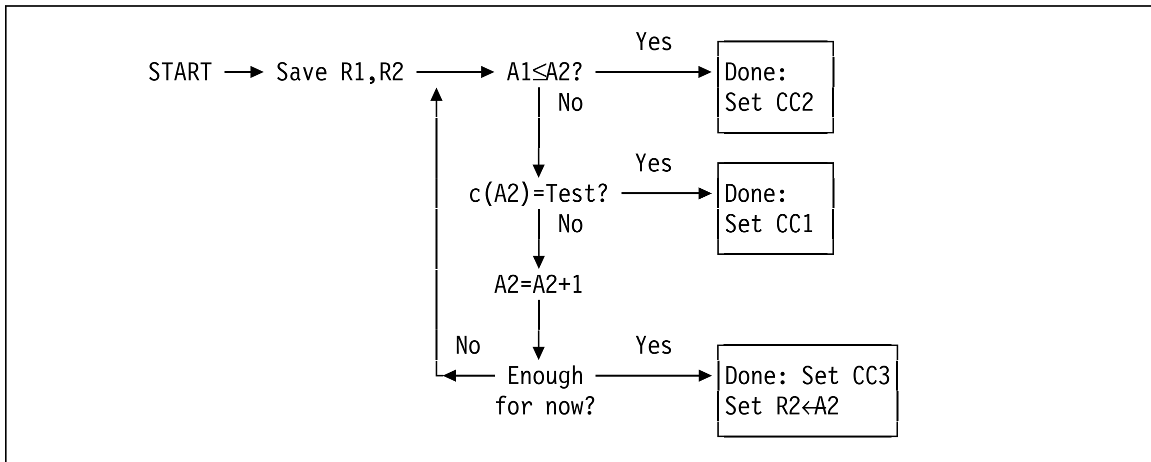


Figure 235. Conceptual execution of the SRST instruction

If the test character is found at the moment the CPU has "scanned enough for now" and would otherwise set the CC to 3, it may instead set the CC to 2; the net result is the same because branching back to the SRST instruction when CC=3 will immediately produce CC=2.

For example, suppose you want to scan the string at **MyData** to find the first occurrence of a blank character:

```
         LA    0,C' '              Search character is a blank
         LA    1,MyData            Set GR1 to start of the string
         LA    5,MyData+L'MyData   Set GR5 to byte past end of string
Repeat   SRST  5,1                 Scan the string for a blank
         JO    Repeat              Scan was incomplete, try again
         JH    NotFound            CC2, no blank was found
         - - -                     GR5 points to the blank
```

If the Condition Code is 3, we simply branch back to the SRST to continue the search.

Exercises

25.4.1.(2) Write a sequence of instructions to find the last nonblank character in a C-string of characters stored at **CData**, whose length (at most 256 bytes) is stored in the word at **CDataLen**. If all blank characters are found, branch to **AllBlank**. If the C-string is empty, branch to **NullData**.

25.4.2.(2)+ The C/C++ programming languages define the **strlen** function to return the length of a C-string argument. Suppose a C-string of unknown length is stored at **WorkArea**. Store its length in the word at **WorkLen**.

25.4.3.(3) The C/C++ programming language function **memchr** searches the first N bytes of a C-string argument to find the first occurrence of a given byte. Suppose a C-string of unknown length is stored at **WorkArea**, and you want to find an occurrence in the string of the byte stored at **FindByte**, and the maximum number of bytes to search is stored in the word at **N**. If the desired character is found, put its address in GR1; if not found, set GR1 to zero.

25.4.4.(2) A single byte is stored at **OddByte**. Write instructions to search for its first occurrence in the C-string stored at **Clutter**. If found, set GR9 to the address of the first occurrence; if not found, set GR9 to zero.

25.4.5.(3) Suppose your program processes words and sentences, and you must alternately search for the blank ending a word and a nonblank starting the next word. Write a sequence of instructions that show how to scan a string at **TextLine** and build arrays containing (a) the length of each word, and (b) its starting address.

25.4.6.(4) Repeat Exercise 25.4.5 but assume that the words might be followed by punctuation characters that should not be stored as part of the word.

25.5. Move String Instruction

The MVST instruction moves bytes from the second operand to the first, testing each source byte for the ending character in the rightmost byte of GR0. If the entire operand (including the ending character) has been moved, the CPU sets Condition Code 1, and sets the R_1 register to the address of the ending character. If some bytes remain to be moved, the addresses in R_1 and R_2 are updated to point to the next bytes to be processed, unused high-order addressing bits are set to zero, and the Condition Code is set to 3. Destructive overlap is not recognized, so be careful!

CC	Meaning
1	Entire second operand moved; R_1 points to end of first operand
3	Incomplete move; R_1 and R_2 point to next bytes to process

Table 161. CC settings for MVST instruction

Figure 236 on page 418 sketches the operation of the MVST instruction. The notation is the same as in Figure 235 on page 416.

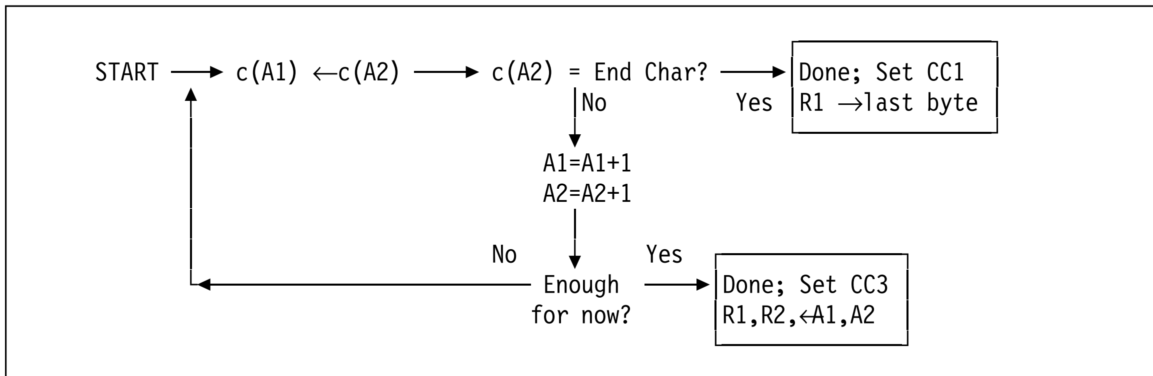


Figure 236. Conceptual execution of the MVST instruction

The following example moves a null-terminated string from **Old** to **New**.

```
         XR    0,0               Ending character is a null byte
         LA    1,Old             Address of source string
         LA    2,New             Address of target string
Move     MVST  2,1               Move from Old to New
         JO    Move              Repeat if CC=3
         - - -
Old      DC    C'This is a null-terminated string',X'0'
New      DS    CL(L'Old+1)       Reserve space for New string
```

Figure 237. Moving a null-terminated string

It's important to ensure that the target field is long enough to hold both the characters and the null terminating byte.

Many programs must scan character strings containing tokens separated by commas. Using MVST, you can move the tokens one at a time to a work area for analysis.

```
         LHI   0,C','            Ending character is a comma
         LA    1,Source          Address of source string
NextTok  LA    2,WorkArea        Address of work area for a token
         LR    3,1               Save starting address of token
Move     MVST  2,1               Move from Source to WorkArea
         JO    Move              Repeat if CC=3
         SR    1,3               Subtract token's starting address
         STH   1,TokenLen        Save its length
         - - -                   Process the token (preserve GR0,GR3)
         LA    1,1(1,3)          Point GR1 past the comma
         J     NextTok           And go scan for the next token
         - - -
Source   DC    C'LIST,OBJECT,XREF,ADATA,'   String of tokens
WorkArea DS    CL20              Reserve space for longest token
TokenLen DS    H                 Length of current token
```

Figure 238. Using MVST to isolate comma-separated tokens

This example is incomplete because we would expect more tokens to follow the last one (ADATA), and because the length of the entire string should be checked to see if the last token was not followed by a comma.

Exercises

25.5.1.(2)+ What would happen in Figure 237 if you used an MVC instruction to move the string from **Old** to **New**? (Assume the string is less than 250 bytes long.)

25.5.2.(2) In Figure 238 on page 418, how would you know that you had correctly processed the last token in the source string?

25.5.3.(2) In Figure 238 on page 418, is the length stored at **TokenLen** the length of the token, or the length of the token and its terminating comma?

25.5.4.(3) Suppose a C-string is stored at **From** and you want to move it to **Target** but with the additional limitation that at most N bytes are moved, where N is stored at **NBytes**. If the null character terminating the string at **From** is moved, set GR1 to the address of the byte following the null character; if the null character is not moved, set GR1 to zero.

25.5.5.(2)+ Write instructions to concatenate the C-string stored at **Suffix** at the end of the C-string stored at **Prefix**. Make sure that the resulting C-string is terminated correctly. Assume that the string at **Suffix** is at most 8000 bytes long.

25.5.6.(3) Repeat Exercise 25.5.5, but assume that the amount of space available for the concatenated string is only 150 bytes. If the result will not fit in the space available, branch to **TooLong**.

25.5.7.(2)+ Write instructions to copy a C-string from **Here** to **There**.

25.5.8.(3) Modify the instructions in Figure 238 on page 418 to scan and process all the tokens in the character string, given that the total length of the token string is in a word at **StrLen** and that the string might contain only a single token without a trailing comma.

25.6. Compare Logical String Instruction

As we saw for CLCL and CLCLE, the two operands being compared can have different lengths, and either operand may be padded. CLST, however, requires that the operands have the same terminating character; and neither is padded during comparison.

The operands are compared byte by byte from left to right, until unequal bytes are found or the end of an operand is reached. Unlike SRST, there is no *stop address* or operand length for either operand, so be sure the strings are properly terminated.

If the end character is found in either operand before being found in the other, the shorter operand is low; if they are found at the same time, the operands are equal.

The Condition Code settings after CLST are the same as those for other compare instructions, except that CC3 indicates an incomplete operation. As with SRST and MVST, if the Condition Code is 3, you can just branch back to repeat the CLST.

CC	Meaning
0	Entire operands are equal; R_1 and R_2 unchanged
1	First operand low; R_1 and R_2 point to last bytes processed
2	First operand high; R_1 and R_2 point to last bytes processed
3	Operands equal so far; R_1 and R_2 point to next bytes to process

Table 162. CC settings for CLST instruction

Figure 239 on page 420 sketches the operation of the CLST instruction. The notation used in the figure is:

A1 Address of a first-operand byte, c(R_1)
c(A1) The first-operand byte at address A1
A2 Address of a second-operand byte, c(R_2)
c(A2) The second-operand byte at address A2
End End character in GR0
x:y x is compared to y

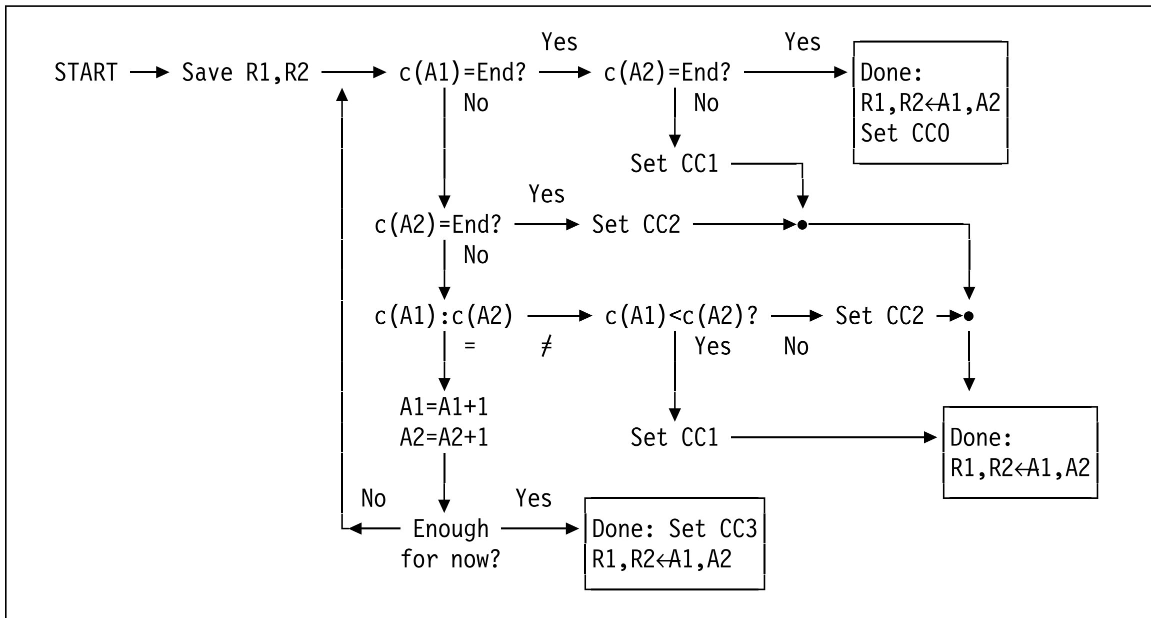


Figure 239. Conceptual execution of the CLST instruction

At termination, the contents of the registers are:

1. If the comparison ends at the End character for both operands, they are equal: the CC is set to 0 and GR R_1 and GR R_2 are unchanged.
2. If the comparison ends at unequal bytes, the CC is set to 1 or 2 depending on whether the first operand byte is less than or greater than the second operand byte. GR R_1 and GR R_2 contain the addresses of the unequal bytes.
3. If the comparison reaches the End character of one operand before the other, that operand is considered the smaller and the CC is set accordingly. GR R_1 and GR R_2 point to the bytes where the comparison stopped. (Note that no padding occurs!).
4. If the comparison is not complete when the CPU needs to allow for possible interrupts, the CC is set to 3 and GR R_1 and GR R_2 have the addresses of the next bytes to be compared. You can then branch back to the CLST instruction to continue comparing.

To illustrate, suppose you want to compare the two C-strings at **A** and **B**.

```
         XR    0,0                Set null ending byte in GR0
         LA    7,A                Set GR7 to start of first operand
         LA    5,B                Set GR5 to start of second operand
Comp     CLST  7,5                Compare the two strings
         JO    Comp               Incomplete comparison, repeat
         JE    Equal              Strings are equal
         JH    A_High             String A compares higher than string B
         J     A_Low              String A compares lower than string B
```

When an inequality is found, the ending characters of the operands are *not* part of the comparison. However, when R_1 and R_2 are updated when the Condition Code is 3, they could contain the addresses of either or both ending characters.

Exercises

25.6.1.(2) Write an instruction sequence that will compare the C-string stored at **StringA** to the C-string stored at **StringB**. Set GR0 to +1 if StringA is greater than StringB, to zero if they are equal, and to −1 if c(StringA) is less than c(StringB).

25.6.2.(3) Write an instruction sequence that will compare the first N bytes of the C-strings stored at **StringA** and **StringB** respectively, where the number of bytes N is stored in the word at **NBytes**. Set GR0 to +1 if StringA is greater than StringB, to zero if they are equal, and to

–1 if StringA is less than StringB. Be sure to handle cases where either or both strings are shorter than N bytes.

25.6.3.(2)+ Suppose these instructions are used to compare two C-strings:

```
         XR    0,0               Ending character is a null byte
         LA    1,X               First operand is at X
         LA    2,Y               Second operand is at Y
         CLST  1,2               Compare first and second operands
```

For each of the following, assume that the string named **X** is at address X'26F943'. Give the Condition Code setting and the addresses in GR1 and GR2 after comparing each pair of strings.

```
(1)      X        DC    C'ABCD',X'0'
         Y        DC    C'ABJE',X'0'

(2)      X        DC    X'0'
         Y        DC    X'0'

(3)      X        DC    C'ABCD',X'0'
         Y        DC    C'ABCDEFGH',X'0'

(4)      X        DC    C'BCDEFGH',X'0'
         X        DC    C'ABCD',X'0'
```

25.6.4.(4) Two strings of bytes begin at **A** and **B** and their lengths are stored in the halfwords at **LA** and **LB** respectively. Compare the two strings up to the length of the shorter; however, if a mismatch occurs and one of the unmatched bytes is X'FF', continue comparing. (Thus X'FF' is a "don't care" character which can match any other character.) Branch to **AB_Equal**, **A_High**, or **B_High** accordingly.

25.7. Translate Extended Instruction

The TRE instruction is similar in function to TR. In both cases, the first operand is the string of bytes to be translated, scanning from left to right, and the second operand is the translate table. There are several differences:

- Addresses: specified in base-displacement form for TR, but in R_1 (which must be even) and R_2 for TRE.
- Lengths: for TR, encoded in the instruction itself, but in R_1+1 for TRE.
- Stop condition: for TR, all first operand bytes are translated; for TRE, either all first operand bytes are translated, or a first operand byte matches the "stop" character in GR0.
- Condition Code: unchanged by TR, but updated by TRE.
- Operand overlap: TR operates byte-by-byte so that operand overlap has no effect; for TRE the results are unpredictable.

One important result of having a stop character is that it can't be translated, unless you add extra instructions to do your own "translation" after TRE completes.

Table 163 gives the Condition Code settings following execution of a TRE instruction:

CC	Meaning
0	All bytes translated; R_1 incremented by length, R_1+1 set to 0
1	R_1 points to the byte matching the stop character; R_1+1 decremented by the number of bytes processed before the match
3	R_1 incremented and R_1+1 decremented by the number of bytes processed

Table 163. CC settings for TRE instruction

To illustrate, suppose a sentence of text starting at **Sentence** is known to be at most 800 bytes long. We want to translate all alphabetic characters to upper case, and stop on the first period.

```
         LHI   0,C'.'             Stop character in GR0
         LA    1,UpperTbl         Address of translate table
         LA    2,Sentence         String to be translated
         LHI   3,800              Maximum length
UpChars  TRE   2,1                Translate characters to upper case
         JO    UpChars            Repeat if not finished
         JZ    NoPeriod           All characters translated, but ...
*                                 ... no stop character was found.
         - - -                    GR2 has address of the stop character
```

Figure 240. Translating characters to upper case with TRE

If we need to translate additional text, we can simply increment GR2, reset the length in GR3, and continue.

The translate table referenced in Figure 240 could be defined with statements like these:

```
UpperTbl DC    256AL1(*-UpperTbl) Initialize table to identities
         Org   UpperTbl+C'a'      Position at C'a'
         DC    C'ABCDEFGHI'       Upper-case equivalents
         Org   UpperTbl+C'j'      Position at C'j'
         DC    C'JKLMNOPQR'       Upper-case equivalents
         Org   UpperTbl+C's'      Position at C's'
         DC    C'STUVWXYZ'        Upper-case equivalents
         Org   ,                  Reposition Location Counter
```

Exercises

25.7.1.(2) What do you think will happen to a TRE instruction if $R_1 = 0$ or $R_2 = 0$? If $R_1 = R_2$?

25.7.2.(2)+ Suppose you execute these instructions:

```
LA    0,C'?'
LHI   3,N
LA    2,X
LA    9,Table
TRE   2,9
```

Assuming an appropriate translate table has been defined at **Table**, show the contents of GR2, GR3, and the Condition Code for each of the following values of N and byte strings starting at **X** which is at address X'7F290C'.

```
(1)      N    Equu    7
         X    DC     C'Who? What?'

(2)      N    Equ    7
         X    DC     C'Unknown?'

(3)      N    Equ    50
         X    DC     10C'Possibly? '
```

25.8. Compare Until Substring Equal Instruction (*)

CUSE is a very complex instruction.[157] It is unusual in another way: it is both interruptible ("Method A") and stops and sets Condition Code 3 to allow interruption processing ("Method B").[158] Though not widely used, it may be applicable in certain applications.

Op	Mnem	Type	Instruction
B257	CUSE	RRE	Compare Until Substring Equal

Table 164. Compare Until Substring Equal instruction

In general, there are two types of matching substring, depending on whether the equal substrings are at the same or different offsets:

- In `'XBCY'` and `'ABCD'`, the equal substrings `'B'` and `'BC'` (with lengths 1 and 2 respectively) are at offset 1.
- In `'XYBC'` and `'ABCD'`, the equal substrings `'B'` and `'BC'` are at different offsets.

The CUSE instruction searches only for equal substrings at the *same* offset, and having the length specified in GR0. It requires six general registers, two of which are fixed: GR0 and GR1. The rightmost byte of GR0 contains the length of the desired matching substrings, and the rightmost byte of GR1 contains a padding byte. The remaining bits of both registers are ignored.

The addresses of the two operands are specified by the even-numbered registers R_1 and R_2, and their lengths are in R_1+1 and R_2+1, respectively. And unlike instructions like MVCL and CLCLE, the lengths are *signed*, and a negative length is treated as zero.[159]

It's important to remember that the substrings must occur at the *same* offset in both operands. Thus, in the two strings

```
ABCDEFG         and         QRSDEFT
```

the substring `DEF` occurs at offset 3, so CUSE can identify matching substrings for lengths 1, 2, and 3. However, in the two strings

```
ABCDEFG         and         BCDEFGH
```

the string `BCDEFG` appears at different offsets, so they will not be considered as equal substrings by CUSE.

The padding character in GR1 is used to extend the shorter string if necessary. For example, if the padding byte is `C'*'` and the two operand strings are

```
ABC             and         BCD**
```

with lengths 3 and 5 respectively, and the substring length is 2, then the matching substring will be the characters **.

The Condition Code and registers are set as indicated in Table 165 on page 424.

---

157 Other complex instructions include EDIT and EDMK; we'll see them in Section 30 when we describe packed decimal arithmetic.

158 At the time of this writing, I know of no other instruction that supports both types of interruption management.

159 A signed length seems strange, as it's hard to think of uses for strings with negative lengths. Other instructions like MVCL and MVCLE use unsigned lengths. (See Exercise 25.8.7.)

CC	Meaning
0	Equal substrings found; R_1, R_2, and lengths updated; or, the substring length is 0, and R_1, R_2 are unchanged
1	Ended at longer operand, last bytes were equal (allows continuing search for further matches if required)
2	Ended at longer operand, last bytes were unequal; or, both operand lengths = 0 and the substring length is > 0
3	Search operation incomplete, last compared bytes unequal; R_1 and R_2 and lengths are updated

Table 165. Condition Code settings by CUSE

Here are some examples of CUSE: suppose we execute the code sequence in Figure 241 for various values of **String1** and **String2** and their lengths, with different pad characters, searching for matching 3-byte substrings in each case:

```
LA    0,Substr_Len       Desired substring length in R0
LA    1,Pad_Char         Pad Character in R1
LM    2,5,=A(String1,L'String1,String2,L'String2)
CUSE  2,4
```

Figure 241. Examples using the CUSE instruction

The results are shown in Table 166; matching substrings are underlined.

String1	L1	String2	L2	Substr Len	Pad Char	CC	L1 after	L2 after	Result
CABCEFDEFEAB	12	ACBABBCEFEAB	12	3	C' '	0	5	5	Match
ABCDEF	6	BCDEFA	6	3	C' '	2	0	0	No match
ABCBACAC	9	BCBABCAC	9	3	C' '	0	3	3	Match at end
ABC	3	CABAAA	6	3	C'A'	0	0	3	Match with pad
ABC	3	CABCAB	6	3	C'A'	2	0	0	No match
ABCBA	5	BCBAA	5	3	C'A'	1	1	1	No match, last bytes equal

Table 166. Results of examples using the CUSE instruction

Searching for matching substrings can be a complex and tedious process, especially if different offsets are allowed. (See Exercise 25.8.1 and Programming Problem 25.1.)

Exercises

25.8.1.(4) Write a sequence of instructions using CLCLE instructions to emulate the function of CUSE.

25.8.2.(2)+ What is the length of the longest matching substring that can be found using CUSE?

25.8.3.(2) Suppose a CUSE instruction detects an inequality following several equal bytes, but the number of equal bytes is less than the required substring length. Should the instruction restart its comparison at the second equal bytes, or at the bytes following the inequality?

25.8.4.(2)+ Suppose your CUSE instruction specifies a substring length 2 with padding character A. If the strings ABCA and DEFA are compared, will it find a matching substring AA?

25.8.5.(2) If the substring length is 1, how is CUSE similar to and different from CLCLE?

25.8.6.(4) Create a flow diagram for CUSE, similar to those in Figures 236 and 239.

25.8.7.(5) Suppose the CUSE instruction supports negative operand lengths, and performs a *backward* search. For example, if **StringA** is ABCD and has length +4, while **StringB** is WCBZ and

has length −4. If the search starts at the rightmost byte of an operand with negative length, it would find a matching substring BC in this case.

Write instructions to emulate a CUSE instruction that supports negative operand lengths.

25.9. Summary

Null-terminated C-strings must be handled carefully. If the terminating null byte is omitted, programs scanning or moving such strings may process far more data than intended, possibly overwriting other data or parts of the program.

The instructions discussed in this section are listed in Table 167; all set the Condition Code.

Function	Length control	End-char control
Move	MVCL MVCLE	MVST
Compare	CLCL CLCLE CUSE	CLST
Search		SRST
Translate	TRE	

Table 167. Extended instructions for character data

Exercises

25.9.1.(3)+ The C/C++ function strncpy copies at most N characters from a C-string at **From** to a C-string at **To** and pads it with null bytes if the "From" string has fewer than N characters. Assuming that the number N is stored in a word at **NBytes**, write an instruction sequence to perform this function.

25.9.2.(2)+ The C/C++ function strcat concatenates characters from the C-string at **Second** to the end of the C-string at **First**. Write an instruction sequence to perform this function, being sure that the result has only a single null character.

25.9.3.(3) The C/C++ function strncat concatenates at most N characters from a C-string at **Second** to the end of the C-string at **First** and terminates the result with a null byte. Assuming that the number N is stored in a word at **NBytes**, write an instruction sequence to perform this function.

25.9.4.(3) The C/C++ function strncmp compares at most N characters from the C-string at **A** to the C-string at **B**. Assuming that the number N is stored in a word at **NBytes**, write an instruction sequence to perform this function, setting GR0 to +1 if A>B, to 0 if A=B, and to −1 if A<B.

25.9.5.(2) The C/C++ function strchr searches a C-string for the first occurrence of a character. Write instructions to perform this function, assuming that the C-string is stored at **CString** and the character to be sought is stored at **FindChar**. If the character is found, set GR3 to its address; otherwise, set GR3 to zero.

25.9.6.(3)+ The C/C++ function strrchr searches a C-string for the *last* occurrence of a character. Write instructions to perform this function, assuming that the C-string is stored at **CString** and the character to be sought is stored at **FindChar**. If the character is found, set GR3 to its address; otherwise, set GR3 to zero.

25.9.7.(3) The C/C++ function strspn searches a C-string for any of the characters in a second C-string, and returns the length of the initial portion of the first string containing characters belonging to the second. Assuming that the C-strings are stored at **First** and **Second**, write instructions that will place in GR1 the length of the first part of the first string containing only characters from the second.

25.9.8.(3) The C/C++ function strcspn searches a C-string for any of the characters *not* in a second C-string, and returns the length of the initial portion of the first string containing no characters belonging to the second. Assuming that the C-strings are stored at **First** and **Second**, write instructions that will place in GR1 the length of the first part of the first string containing none of the characters from the second.

25.9.9.(3) The C/C++ function strpbrk searches a C-string for the first occurrence of *any* character in a second C-string, and returns the address of the character if present, or a null (zero) pointer if none is found. Assuming that the C-strings are stored at **First** and **Second**, write instructions that will place in GR1 the address of the first occurrence in the first string of any character from the second, or zero if none is found.

25.9.10.(3) The C/C++ function strstr searches a C-string for the first occurrence of a second C-string, and returns the address of the first character of that matching string if present, or a null (zero) pointer if none is found. Assuming that the C-strings are stored at **First** and **Second**, write instructions that will place in GR1 the address of the first occurrence in the first string of the second string, or zero if none is found. If the second string is null, return the address of the first.

25.9.11.(2) The C/C++ function memchr searches N bytes in memory for the first occurrence of a character, and returns a pointer to the character if present or a null (zero) pointer if none is present. Write instructions to perform this function, assuming that the data to be searched is stored at **MemData**, the character to be sought is stored at **FindChar**, and the number N is stored in a word at **NBytes**. If the character is found, set GR3 to its address; otherwise, set GR3 to zero.

25.9.12.(2) The C/C++ function memset stores a character into the first N bytes of a C-string. Write instructions to perform this function, assuming that the C-string is stored at **CString**, the character to be stored is at **FillChar**, and the number N is stored in a word at **NBytes**.

What will happen if the null bytes at the end of the C-string is overwritten, or if no null byte is placed after the N-th byte?

25.9.13.(4)+ A string of EBCDIC characters starting at Str+2 contains substrings of blanks and nonblanks. The total length of the string is a halfword binary integer in the two bytes at Str. Write instructions to replace multiple blanks in the string with a single blank, and update the string length accordingly. (Such a result is sometimes called “blank-compressed”.)

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
CLCL	0F
CLCLE	A9
CLST	B25D

Mnemonic	Opcode
CUSE	B257
MVCL	0E
MVCLE	A8

Mnemonic	Opcode
MVST	B255
SRST	B25E
TRE	B2A5

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
0E	MVCL
0F	CLCL
A8	MVCLE

Opcode	Mnemonic
A9	CLCLE
B255	MVST
B257	CUSE

Opcode	Mnemonic
B25D	CLST
B25E	SRST
B2A5	TRE

Terms and Definitions

C-string
: A string of zero or more bytes ending with a zero or “null” byte.

destructive overlap
: Destructive overlap occurs when any part of a target operand field is used for a source after data has been moved into it.

interruptible
: An instruction is interruptible if the CPU suspends its operation, updates the registers involved in the operation and subtracts the instruction's length from the Instruction Address in the PSW, so that when the program resumes execution, the instruction will start from the point where it was interrupted.

null byte
: A zero or `X'00'` byte, sometimes indicated by the character ***n***.

Programming Problems

Problem 25.1.(3) Write a program that reads two character strings from two 80-byte records, and searches for the first and longest matching substring at *any* offset within the two strings. Use a blank for the padding character. Print the original strings, the matching substring and its length, and its offset within each string. Repeat for several pairs of input strings.

For example, if the two strings are `'XYA12345'` and `'$12345678'`, the longest matching substring is `'12345'` with length 5, at offsets 3 and 1 respectively. The requirement that you find the longest matching substring means that you shouldn't report a one-byte substring like `'1'`.

Problem 25.2.(3)+ Programs must sometimes isolate a string of characters preceded and followed by strings of blanks. For example, if the original string is `'•••AB•CD••'` (where • means a blank character), the desired result is the string `'AB•CD'`.

Write a program that reads 80-character records and removes leading and trailing blanks. Print the original record, and the “blank-trimmed” result and its length. Repeat for several input records.

Some sample input records might include

```
         DC    CL80'   AB CD  '    As in the example above
         DC    80C'*'              No blanks
         DC    CL80'AB'            No leading blanks
         DC    CL78' ',C'YZ'       No trailing blanks
         DC    CL80' '             All blanks
```

You should create other records to exercise your program.

26. Other Types of Character Data (*)

```
 2222222222    6666666666
222222222222  666666666666
22        22  66        66
          22  66
          22  66
         22   66666666666
       22     666666666666
     22       66        66
   22         66        66
 22           66        66
222222222222  666666666666
222222222222   6666666666
```

For many programs, you need not be concerned with the details of the character representation used for your programs. There are times, however, when programs need to recognize the encodings used for data they read or create. This section introduces some of the other character encodings you may meet, and instructions to help process them.

26.1. Character Representations

Data is stored as strings of bits; we interpret the meanings of the bits differently for different data types. Every computer system must solve the problem of representing characters as bit patterns.

Many character encodings exist, and many others have been forgotten. As the number of encodings for a given character grew it became much more difficult to exchange data among systems. This led to efforts to standardize on a smaller set of codes; among them are single-byte EBCDIC, ASCII, double-byte EBCDIC, and Unicode. We'll investigate each in turn.

26.1.0. An Early Character Encoding

On the earliest "Institute" machines, input and output were done with punched paper tape prepared and printed on Teletype™ machines. The characters were encoded as five bits on tapes with six holes, a smaller one of which was a "feed" or "sprocket" hole just above the center of the tape, that was used to move the tape on mechanical readers or measure its progress on photoelectric cell readers. An example of a tape segment is shown in Figure 242:

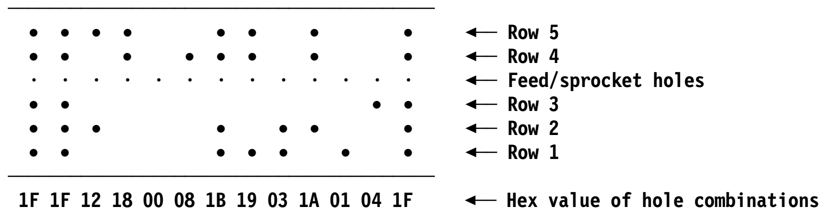


Figure 242. Fragment of an Institute-machine punched paper tape

Because there were only five data-bit positions on the tape, encoding decimal digits, capital letters, and special characters required switching between "number shift" and "letter shift". This means that each combination of five bits might mean two different things, depending on which shift

mode was active (much as on a typewriter keyboard with a shift lock). These bit combinations are shown in the following two tables.[160]

Hex Value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Row 5																
Row 4									•	•	•	•	•	•	•	•
Feed hole	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Row 3					•	•	•	•					•	•	•	•
Row 2			•	•			•	•			•	•			•	•
Row 1		•		•		•		•		•		•		•		•
Num Shift	0	1	2	3	4	5	6	7	8	9	+	–	N	J	F	L
Ltr Shift	P	Q	W	E	R	T	Y	U	I	O	K	S	N	J	F	L

Table 168. Punched paper tape encodings with values 00-0F

Hex Value	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Row 5	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Row 4									•	•	•	•	•	•	•	•
Feed hole	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Row 3					•	•	•	•					•	•	•	•
Row 2			•	•			•	•			•	•			•	•
Row 1		•		•		•		•		•		•		•		•
Num Shift	Dly	$	CRLF	(	Ltr Shf	,	)	/	Dly	=	.	Num Shf	'	:	*	Spc
Ltr Shift	Dly	D	CRLF	B	Ltr Shf	V	A	X	Dly	G	M	Num Shf	H	C	Z	Spc

Table 169. Punched paper tape encodings with values 10-1F

In Table 169, “Dly” means “Delay”,[161] “CRLF” means “Carriage Return and Line Feed”, “Ltr Shf” means “Letter Shift”, “Num Shf” means “Number Shift”, and “Spc” means “Space”.

Normal tape input skipped characters with a hole in Row 5; special instructions were used to read characters using any combination of holes.

Note that you can't start reading a tape at any arbitrary position, because you won't know which shift mode is active. (We will see in Section 26.4 that there is a similar problem with Double-Byte EBCDIC character sets.)

26.1.1. BCD characters

The EBCDIC code used in System z is a descendant of the older BCD (“Binary Coded Decimal”) punched-card encodings used on many early IBM processors. Each character was 6 bits wide, and was represented as two octal digits.[162] Other computer manufacturers use similar (and sometimes very different) six-bit character encodings.

Table 170 on page 430 gives the BCD encodings. There are no lower-case characters, and fewer “special” characters than are supported by EBCDIC. (See Exercise 26.1.3.)

[160] These punched paper tape codes were used on the “ILLIAC 1” machine at the University of Illinois, the first “Institute” machine at a university. Many other five-bit teletype codes were used worldwide.

[161] The “Delay” code was needed to allow time for the teletype printer carriage to return to the left margin before printing the next characters.

[162] We mentioned base-8 (octal) encoding in Section 2.2.

Octal	Char	Octal	Char	Octal	Char	Octal	Char
00	0	20	+ &	40	–	60	blank
01	1	21	A	41	J	61	/
02	2	22	B	42	K	62	S
03	3	23	C	43	L	63	T
04	4	24	D	44	M	64	U
05	5	25	E	45	N	65	V
06	6	26	F	46	O	66	W
07	7	27	G	47	P	67	X
10	8	30	H	50	Q	70	Y
11	9	31	I	51	R	71	Z
13	= #	33	.	53	$	73	,
14	' @	34	) ¤	54	*	74	(%

Table 170. Old six-bit BCD character representation

Table entries with more than one character show variations in the BCD character representations used among different IBM processors.[163]

Exercises

26.1.1.(1) Suppose a string of 60 bytes contains characters encoded in BCD; each byte has two high-order zero bits. Write a sequence of instructions that will convert the BCD characters to EBCDIC.

26.1.2.(2)+ Suppose a string of 72 bytes contains 96 BCD six-bit characters packed together with no padding bits. That is, each group of three bytes contains four BCD characters. Write a sequence of instructions that will “unpack” the BCD characters from the string starting at **BCDChars** into a string of 96 EBCDIC characters starting at **EBCDChar**. Translate characters as needed.

26.1.3.(1) If the card image in Table 12 on page 78 uses the BCD coding in Table 170, what 40 octal digits will encode the characters in the first 20 columns?

26.1.4.(1) Assuming that punched paper tapes are prepared starting in Letter-shift mode, what are the characters on the Teletype tape in Figure 242 on page 428? What are the characters if the tape is prepared starting in Number-shift mode?

26.1.5.(1) In Figure 242 on page 428, a hole punched in Row N corresponds to what power of 2?

26.2. EBCDIC Representations and Code Pages

In System/360 processors, the 8-bit byte required a new 8-bit character encoding, EBCDIC. Initially, the character set included mainly the characters used in the United States.

However, these early processors were soon used in many countries where the EBCDIC characters shown in Table 13 on page 87 did not provide “national” characters. For example, “father” in French (“père”) requires e-grave, and “young woman” in German (“Fraülein”) requires u-umlaut.

Thus, additional EBCDIC encodings were created; these are grouped in “code pages” giving the characters assigned to each of 256 possible bit patterns. It was difficult to exchange data and

[163] The ¤ symbol (encoding 34) was used as a substitute for the many currency symbols used in different countries.

programs because the same character encoding was often different among code pages. For example, Table 171 on page 431 shows some of the varying hexadecimal encodings used in current EBCDIC tables, for these language groupings:

Code Page	**Description**
037	Original EBCDIC: this "ancestral" EBCDIC code page is the code page used for Assembler Language programs
500	Modern EBCDIC; a few special characters have different encodings than in code page 037
1047	International-1
1140	United States, Canada, Netherlands, Australia, New Zealand, Portugal, Brazil
1141	Austria, Germany
1142	Denmark, Norway
1143	Finland, Sweden
1144	Italy
1145	Spain, Latin American Spanish
1146	United Kingdom
1147	France
1148	International-1, supporting most "Western" languages
1149	Iceland
1153	Eastern Europe

All the code pages above 1140 support the euro character ∊.[164]

Table 171 shows some examples of why you can't assume that a given encoding represents an expected character. As your program is moved from region to region (and displayed or printed on unknown code pages), many of the encodings for the characters are different.

Char	037	500	1047	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1153
$	5B	5B	5B	5B	5B	67	67	5B	5B	4A	5B	5B	5B	5B
@	7C	7C	7C	7C	B5	80	EC	B5	7C	7C	44	7C	AC	7C
#	7B	7B	7B	7B	7B	4A	63	B1	69	7B	B1	7B	7B	7B
|	4F	BB	4F	4F	BB	BB	BB	BB	4F	4F	BB	BB	BB	6A
¬	5F	BA	B0	5F	BA	BA	BA	BA	5F	5F	BA	BA	BA	—
[	BA	4A	AD	BA	63	9E	B5	90	4A	B1	90	4A	AE	4A
]	BB	5A	BD	BB	FC	9F	9F	51	5A	BB	B5	5A	9E	5A
{	C0	C0	C0	C0	43	9C	43	44	C0	C0	51	C0	8E	C0
}	D0	D0	D0	D0	DC	47	47	54	D0	D0	54	D0	9C	D0
∊	—	—	—	9F	9F	5A	5A	9F	9F	9F	9F	9F	9F	9F

Table 171. Sample EBCDIC characters with varying code points among code pages

If you write programs or create data that might be sent to other countries, it helps to remember that many EBCDIC characters have the same encoding across almost all modern EBCDIC code pages. These "invariant" characters are called the *Syntactic Character Set*:

- blank/space
- decimal digits
- upper-case and lower-case Latin-alphabet letters
- these special characters:

```
+ < = > % & * " ' ( ) , _ - . / : ; ?
```

Using just these 82 characters will also help you avoid the possibility that your program could be difficult to read when printed, or when displayed on a different terminal or workstation. Note that the three "national" characters allowed in Assembler Language symbols (the dollar sign $, the

164 This is the best available approximation to the euro character in the character set used for this text.

at sign @, and the sharp, hash, octothorpe, or (US) pound sign #) are *not* invariant across EBCDIC code pages!

Exercises

26.2.1.(3) Suppose your program contains character data written using code page 037, and you must translate the character data to the code page used in Sweden, 1143. Write statements to generate a 256-byte table for translating characters in code page 037 to characters in code page 1143. Assume that all the varying characters are those shown in Table 171 on page 431. (For extra credit: try to find if there are any other "variant" characters between those two code pages.)

26.2.2.(1) What does your Assembler generate if you code this statement?

```
         DC    C'@#$|{[]}'
```

26.2.3.(1) How many characters are in the Systactic Character set?

26.3. ASCII

A widely used single-byte character representation is the "American Standard Code for Information Interchange", or ASCII for short. It started as a 7-bit code and was later extended to 8 bits. The hexadecimal character encodings for 7-bit ASCII[165] are shown in Table 172 on page 433.

[165] Different ASCII character encodings are used on other hardware and software systems. This encoding and its 8-bit superset (known as "ASCII Standard 8859-1") is the representation used for System z applications.

Char	Code	Char	Code	Char	Code	Char	Code
blank	20	8	38	P	50	h	68
!	21	9	39	Q	51	i	69
″	22	:	3A	R	52	j	6A
#	23	;	3B	S	53	k	6B
$	24	<	3C	T	54	l	6C
%	25	=	3D	U	55	m	6D
&	26	>	3E	V	56	n	6E
'	27	?	3F	W	57	o	6F
(	28	@	40	X	58	p	70
)	29	A	41	Y	59	q	71
*	2A	B	42	Z	5A	r	72
+	2B	C	43	[	5B	s	73
,	2C	D	44	\	5C	t	74
-	2D	E	45	]	5D	u	75
.	2E	F	46	^	5E	v	76
/	2F	G	47	_	5F	w	77
0	30	H	48	`	60	x	78
1	31	I	49	a	61	y	79
2	32	J	4A	b	62	z	7A
3	33	K	4B	c	63	{	7B
4	34	L	4C	d	64	|	7C
5	35	M	4D	4	65	}	7D
6	36	N	4E	f	66	˜	7E
7	37	O	4F	g	67	(none)	7F

Table 172. 7-bit ASCII character representation

All these characters use only the rightmost 7 bits; the high-order bit is zero. When the high-order bit is 1, more characters are available; many are accented or special. For example, the e-grave character in “père” has ASCII representation X'D8', and the u-umlaut character in “Fraülein” has ASCII representation X'FC'.

The Assembler generates ASCII character constants if you specify subtype A on a C-type constant. For example:

```
ASCIICon DC    CA'ASCII Characters'
```

will generate

```
         X'41534349492043686172616374657273'
*          A S C I I   C h a r a c t e r s
```

The Assembler treats the nominal value of the constant as EBCDIC characters, and translates each byte to ASCII, after pairing of & and ', as for other character constants and terms.

You can also generate ASCII constants by specifying the TRANSLATE(AS) option; this causes all C-type constants (with no subtype) to be translated from EBCDIC to ASCII. Note that character self-defining terms are not translated to ASCII, unless you *also* specify the COMPAT(TRANSDT) option. It is worth remembering that the ASCII code for a blank space is X'20'.

Exercises

26.3.1.(1)+ If you saw a hexadecimal display of a string of alphanumeric characters (letters and digits), what would help you decide whether they were represented in ASCII or EBCDIC?

26.3.2.(2) What machine language data is generated by these statements?

```
Con1     DC    CA'5*(2.236/Denom)+Pi'
Con2     DC    CA'Invalid Expression?'
Con3     DC    CA'Hello, World!'
Con4     DC    CA'BitMask&&Byte||''Chars'''
```

26.4. Double-Byte EBCDIC Data (*)

The EBCDIC characters representable by a single 8-bit byte have too few values to handle ideographic languages like Japanese. To solve this problem, the EBCDIC encodings were extended to encodings with *pairs* of bytes, and two special byte codes allow switching between "double-byte" characters and our familiar single-byte characters.

Single-byte character sets are sometimes abbreviated "SBCS", and a collection of double-byte characters is a "Double Byte Character Set", or "DBCS" for short.

DBCS data is a stream of single bytes, grouped in pairs. Groups of DBCS pairs are always delimited by "Shift-Out" (SO) and "Shift-In" (SI) byte codes:

- Shift-Out (X'0E') shifts *out* of single-byte mode to double-byte mode;
- Shift-In (X'0F') shifts *in* to single-byte mode from double-byte mode.

Data between the SO and SI byte codes is *always* treated as byte pairs. The byte codes for ampersand and apostrophe are not treated specially in DBCS data.

A string of bytes mixing single- and double-byte characters is illustrated in Figure 243, where sb represents a single-byte EBCDIC character, db represents one of the two bytes of a double-byte EBCDIC character, and SO and SI represent the Shift-Out and Shift-In byte codes.

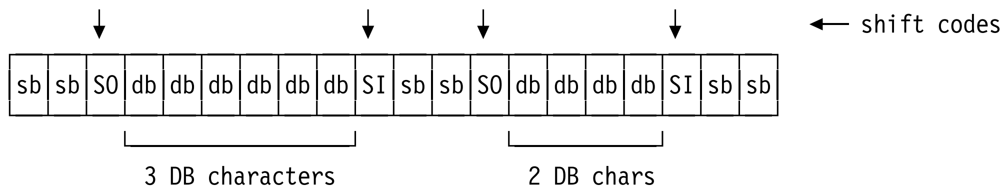


Figure 243. Mixed single- and double-byte EBCDIC characters

These 20 bytes represent 6 single-byte characters and 5 double-byte characters.

Because the byte codes used for SO and SI aren't displayable, we use a special notation for DBCS data.

- It's customary to represent SO pictorially by <, and SI by >. (Remember that the actual characters "<" and ">" are not in the data!) Sometimes the characters ◄ and ► are used instead.
- We represent double-byte characters by Dx. Thus, <Dz> represents a Shift-Out, a double-byte character, and a Shift-In. Its hexadecimal value is represented by 0EDDzz0F. (A confusing notation: the DD characters represent the hex value of the high-order byte of the DBCS pair, and zz represents hexadecimal value of the low-order byte.)
- Latin-alphabet EBCDIC characters in DBCS format are represented by .X, so that the DBCS letter A could be shown as ".A". These are sometimes called "wide" Latin-alphabet characters, because DBCS-sensitive devices display them that way. For example, the DBCS "A" could be written "<.A>".

- Single-byte SBCS EBCDIC characters are represented by "e" (or themselves). Thus, the characters in Figure 243 might be described as "ee<DbDbDb>ee<DbDb>ee".
- Except for DBCS blanks, which have representation X'4040', both DBCS bytes have values in the range [X'41',X'FE']. This is illustrated in Table 174.

Here are some examples of DBCS data:

```
<.A>      X'0E42C10F'       (pure DBCS)
<.'.&>    X'0E427D42500F'   (pure DBCS; & and ' not paired )
<Da>      X'0E....0F'       (whatever Da represents; e.g. X'0EDDaa0F')
<'&>      X'0E7D500F'       (in the user-defined character range)
a<.A>b    X'810E42C10F82'   (mixed SBCS and DBCS data)
```

Figure 244. Examples of DBCS data

Some DBCS EBCDIC code point assignments for Japanese are given in Table 173. The first byte of a DCBS character selects a *ward*, a group of DBCS code points having identical first bytes. Each ward can represent up to 192 DBCS characters.

First (ward) byte	Second byte	Contents
X'41'	X'41-FE'	Greek, Cyrillic, Roman numerals
X'42'	X'41-FE'	Latin alphabets and alphanumerics
X'43'	X'41-FE'	Katakana: phonetic, foreign loan words
X'44'	X'41-FE'	Hiragana: grammatical endings and Japanese indigenous words
X'45-55'	X'41-FE'	Kanji basic set
X'56-68'	X'41-FE'	Kanji extension set
X'69-7F'	X'41-FE'	User definable
X'80-FE'	X'41-FE'	Reserved

Table 173. Japanese DBCS assignments

Thus, the "." in DBCS representations of Latin-alphabet SBCS characters represents the X'42' ward byte. Table 174 shows the overall structure of DBCS character representations; valid DBCS codings are in the shaded areas.

	00	40	41	FF
00				
40				
41				
FF				

Table 174. DBCS encoding

It helps considerably that the "standard" EBCDIC characters shown in Table 13 on page 87 are represented in DBCS with X'42' as the first two digits. That is, if a nonblank character's EBCDIC representation is X'xy', its DBCS representation is X'42xy'.

26.4.1. The DBCS Option (*)

The Assembler's DBCS option controls recognition of DBCS data. It also allows G-type constants and self-defining terms. The byte codes for SO and SI are recognized as shifts only if the DBCS option is active, and ampersand and apostrophe byte codes are not tested for pairing in either byte of a double-byte character.

If the NODBCS option is active (the Assembler's default), normal rules apply:

- Shifts (SO and SI) are just data bytes.
- Nothing is recognized as DBCS data.
- Ampersands and apostrophes between SO/SI are recognized for pairing.
- G-type constants and self-defining terms are not allowed.

26.4.2. G-Type DBCS Constants and Self-Defining Terms (*)

Like any other form of character data, DBCS data must be enclosed in apostrophes. It is allowed wherever EBCDIC character data is allowed.

You can specify both pure and mixed DBCS data in statement operands. Pure DBCS data is DBCS only, as in C'<Da..Dz>'. Mixed DBCS data may contain both single-byte EBCDIC and DBCS data, as in C'ee<Da..Dz>ee'.

Both G-type constants and self-defining terms are written G'dbcs_data', and support only pure DBCS data. The shifts are removed from the generated object code. For example:

```
         DC    G'<DwDxDy>'        Generates X'DDwwDDxxDDyy'  (6 bytes)
         DC    G'<.A.&.B>'                  X'42C1425042C2'  (6 bytes)
```

In addition, redundant SI/SO pairs such as >< are removed, since they have no useful effect in pure DBCS data. For example:

```
         DC    G'<Dx><Dy>'        Generates X'DDxxDDyy'  (4 bytes)
         DC    G'<.A><.B>'                  X'42C142C2'  (4 bytes)
```

Because each DBCS character is two bytes, G-type self-defining terms contain either one or two DBCS characters.

```
DBCSAC   Equu  G'<DxDy>'          Has value  X'DDxxDDyy'
DBCSB    Equ   G'<.B>'            Has value  X'000042C2'
GNull    Equ   G'<>'              Causes an error
```

Constants and terms are padded on the right with DBCS blanks:

```
         DC    GL4'<.B>'          Generates X'42C24040'
```

Explicit lengths must be a multiple of 2 bytes, and truncation is done at the right end of the constant.

C-type constants and self-defining terms may contain either pure DBCS *or* mixed DBCS data. Unlike G-type constants and self-defining terms, the shift bytes *are* generated. For example:

```
         DC    C'1<DxDy>2'        Generates   X'F10EDDxxDDyy0FF2'
         DC    C'<.A.&.B>'                    X'0E42C1425042C20F'
         DC    C'1<Dx>2<Dy>3'                 X'F10EDDxx0FF20EDDyy0FF3'
```

Also, redundant SI/SO pairs (><) are *not* removed. For example:

```
         DC    C'<Dx><Dy>'        Generates   X'0EDDxx0F0EDDyy0F'
         DC    C'<.A><.B>'                    X'0E42C10F0E42C20F'
```

C-type self-defining terms may contain only one DBCS character, because the shift bytes are included:

```
DBCSJ    Equ   C'<.J>'              Has absolute value X'0E42D10F'
CNull    Equ   C'<>'                Value X'00000E0F' (not truly DBCS!)
```

Both G-type and C-type constants are padded on the right with EBCDIC blanks:

```
         DC    CL5'<.B>'            Generates   X'0E42C20F40'
```

and truncation is also on the right. However, truncation into DBCS data not allowed if the DBCS option is specified. For example,

```
         DC    CL3'<.B>'            Generates an error message
```

truncates into the DBCS data.

Thus, identical nominal values are treated differently in G-type and C-type constants. For example:

```
GTerm    Equ   G'<.A>'        Has value   X'000042C1' (no shifts)
CTerm    Equ   C'<.A>'        Has value   X'0E42C10F' (with shifts)
```

26.4.3. Continuation Rules for DBCS Data (*)

When the DBCS option is active, the SO and SI are *not* considered continuation indicators! If an SI appears before the continuation-indicator column at the end of a continued line, and is followed by an SO in the continue column of the next line, they are considered redundant, and are removed in the generated constant.

Because G-type DBCS constants might be displayed on devices sensitive to DBCS characters, DBCS characters should not be split at points of continuation. The Assembler provides a special extended continuation rule when the DBCS option is active, a flexible end column on a line-by-line basis. If the continuation indicator is nonblank, then the end column is the *first* column to the *left* of the continuation indicator that *differs* from the continuation indicator. Assuming default end and continuation columns, Figure 245 shows how extended continuation works:

```
        col.16 ↓                                                col.72 ↓
DBCSA    DC    G'<DaDbDcDdDeDfDgDhDiDjDkDlDmDnDoDpDqDrDsDt>*************
               <DuDvDwDx>///////////////////////////////////////////////
               <DyDzD1D2D3D4D5>'        No SO/SIs in generated constant
*
DBCSB    DC    C'123456789A123456789B123456789C123456789D123456789E=====
               <D1D2D3D4>'              SO/SI are generated
```

Figure 245. Extended continuation for DBCS data

For **DBCSA** the continuation character * at the end of the first line is extended to the left, so the end column becomes the SI code following Dt. And because the continuation line starts with an SO code, both are eliminated. Similarly, the SI following Dx and the SO on the third line are redundant, so the generated constant will be

```
         'DaDbDcDdDeDfDgDhDiDjDkDlDmDnDoDpDqDrDsDtDuDvDwDxDyDzD1D2D3D4D5'
```

However, the constant named **DBCSB** is a C-type constant, so the shift bytes are included, and the generated value also includes four equal sign (X'7E') characters:

```
X'F1F2F3F4F5F6F7F8  F9C1F1F2F3F4F5F6  F7F8F9C2F1F2F3F4  F5F6F7F8F9C3F1F2
  F3F4F5F6F7F8F9C4  F1F2F3F4F5F6F7F8  F9C57E7E7E7E0Exx  F1xxF2xxF3xxF40F'
```

where the xx characters among the last nine bytes are the hexadecimal representation of the high-order byte of the four DBCS characters D1D2D3D4.

Be Careful!

Don't try to process bytes of DBCS data starting at an arbitrary position, because you may not know whether there was a previous Shift-Out or Shift-In byte (or none at all!), or whether your chosen byte is the second byte of a DBCS character.

Exercises

26.4.1.(1) Assuming the DBCS option is specified, what data will be generated by these statements?

```
A        DC    G'<.A.B.C>'
B        DC    GL4'<.A.B.C>'
C        DC    CL4'<.A.B.C>'
D        DC    C'<.A.B.C>'
```

26.5. Unicode

Unicode is a much broader topic than this brief summary can properly describe. We'll first introduce some background, then the character representation, and how to use the Assembler to create Unicode character constants. Section 26.6 will introduce instructions used to process Unicode data.

26.5.1. The Unicode Representation

The growing number of (human) languages using character data led to an international effort to standardize on a single encoding. The resulting Unicode[166] standard has greatly simplified character data exchange among nations and languages. It is identical to the International Standards Organization (ISO) ISO/IEC 10646 standard, and provides for 8-bit (UTF-8), 16-bit (UTF-16), and 32-bit (UTF-32) encodings.[167] All three encodings represent the same repertoire of all characters and (essentially) all languages of the modern world.

Unicode assigns every UTF-16 character a 16-bit numeric “scalar” value, denoted `U+nnnn`, where each `n` represents a hexadecimal digit. (It is the same as `X'nnnn'`.) The 7-bit ASCII character code shown in Table 172 on page 433 is a basic element of the Unicode standard: all 8-bit ASCII codes are used as the 256 lowest UTF-16 values, `U+0000` through `U+00FF`. This is the same as ASCII Standard 8859-1.

Unicode character assignments encompass a truly enormous variety of characters; some subsets are shown (for your amazement) in Table 175 on page 439.[168]

[166] Unicode™ is a trademark of Unicode®, Inc. Unicode is officially known as the Unicode Standard, and was created by a cooperative effort of the Unicode Consortium and the International Standards Organization (ISO). You can find code charts at `http://www.unicode.org/charts/`

[167] UTF is an abbreviation for “Unicode Transformation Format”.

[168] Additional encodings have also been standardized for highly specialized uses such as ancient Phoenecian and Sumero-Akkadian Cuneiform. The *Unicode Standard* contains the encodings and some interesting history.

Range	Block Name	Range	Block Name
U+0000-U+007F	Basic Latin (ASCII)	U+0080-U+00FF	Latin-1 Supplement
U+0100-U+017F	Latin Extended-A	U+0180-U+01FF	Latin Extended-B
U+0250-U+02AF	Phonetics	U+0300-U+0365	Combining Diacritics
U+0370-U+03FF	Greek and Coptic	U+0400-U+04FF	Cyrillic
U+0590-U+05FF	Hebrew	U+0600-U+06FF	Arabic
U+0900-U+097F	Devanagari	U+0980-U+09FF	Bengali
U+13A0-U+13FF	Cherokee	U+1400-U+167F	Canadian Aboriginal
U+2070-U+209F	Superscripts and subscripts	U+20A0-U+20CF	Currency Symbols
U+3040-U+309F	Hiragana	U+30A0-U+30FF	Katakana
U+4E00-U+9FFF	China/Japan/Korea Unified Ideographics	U+E000-U+F8FF	Private Use

Table 175. Sample Unicode assignments

As sometimes happens with international standards, compromises were needed to satisfy the needs of all participants. Originally, Unicode was intended to be a purely 16-bit encoding, but as its popularity grew, more codes were needed to support new characters. This led to a provision for "surrogate" characters: the first 16-bit Unicode character (the "high surrogate") indicates that the next 16 bits (the "low surrogate") are an extension of the the first; together, they are called "surrogate pairs". This allowed including over 1 million new characters without disrupting the basic encoding scheme. (We'll see surrogate pairs again when we describe the format-conversion instructions in Section 26.6.5.) Surrogate pairs are rarely used in Assembler Language programs.

For the next part of this discussion, we'll stay with the UTF-16 encoding, and describe the often-used variant encoding, UTF-8, in Section 26.6.5.

26.5.2. Glyphs and Characters

A character is the basic unit of encoding, and UTF-16 is the most commonly used encoding format. It is important to distinguish between "glyphs"and characters. A glyph is what you see when a character is printed or displayed. For example, displayed forms might include a, *a*, **a**, ***a***, and A, *A*, **A**, ***A***, A. All of these glyphs are representations of the two characters "a" and "A", four in lower case and five in upper case forms (normal, italic, bold, bold italic, and "small caps").

Some "characters" can require more than one 16-bit Unicode character code value. For example, **Ä** has Unicode coding U+00C4 (the same as X'00C4'). Because the base character **A** has been combined or *composed* with an accent, it is called a *pre*composed character. However, **Ä** may also be represented by two *separate* Unicode characters: the **A** (with coding U+0041) and a combining mark ¨ (a diaresis or umlaut, with coding U+0308). This combination is called a *de*composed form.

Similarly, **ñ** (U+00F1) = **n** (U+006E) + ~ (U+0303) represent composed and decomposed forms.

Fortunately, Assembler programs rarely need to handle decomposed characters.

26.5.3. Unicode Character Constants

The Assembler generates Unicode constants if you specify constant type CU: that is, type C with type extension U. The nominal value is a string of 8-bit EBCDIC code points.

Because the same character can have different EBCDIC encodings while Unicode provides unique representations for all characters, the Assembler's CODEPAGE option lets you specify the character representation used for the 8-bit single-byte nominal values in your CU-type constants.

To show why the CODEPAGE option may be important, suppose you live in England. When preparing your program, you enter a byte for the £ "Pound Sterling" character. It has code point X'5B' on code page 1146, the usual code page for the United Kingdom. However, if the CODEPAGE(1146) option is omitted, the Assembler will convert the £ character to U+0024, the Unicode representation of the $ dollar sign! This is because the Assembler assumes a default code

page on which $ has code point X'5B'. When the CODEPAGE(1146) option is specified, the Assembler converts the £ correctly to U+00A3.

These constants are parsed using the usual rules for C-type constants (apostrophe and ampersand pairing). Each byte of the result is then mapped from the EBCDIC encoding specified by the CODEPAGE option to the equivalent 2-byte UTF-16 encoding.

The Length attribute of a CU-type constant is always measured in bytes (not "characters"); an explicit length (if specified) must be even. If the explicit length is longer than the implied length, the byte string is first padded with EBCDIC blanks, and if shorter, the byte string is truncated on the right. Then, it is translated to UTF-16. Implied lengths are the number of bytes generated: 2×(number of EBCDIC characters after pairing).

```
SampleCU DC    CU'Unicode Characters'
```

Figure 246. CU-type constant generating Unicode characters

The statement in Figure 246 generates these 36 bytes from the 18 nominal-value characters:

```
0055006E00690063 006F006400650020 0043006800610072 0061006300740065 00720073
```

Because the characters in the nominal value of the CU-type constant in 246 are members of the invariant EBCDIC character set described on page 431, any of the CODEPAGE option values described on page 431 may be specified to generate the above result.

Notice that each Unicode character starts with nine zero bits (B'0000 0000 0'), meaning that it is representable in the 7-bit portion (U+0000 to U+007F) of the UTF-16 Unicode encoding.[169] The symbol **SampleCU** has Length Attribute 36.

It's important to remember that the Assembler's CODEPAGE option applies *only* when translating EBCDIC characters to Unicode in CU-type constants. The remaining characters in your program are understood by the Assembler to be represented in the 037 code page.

Exercises

26.5.1.(2)+ In Figure 246, what changes to the generated object code would occur if your program had been created using code page 1146?

26.5.2.(4) Search the web to find the character encodings used for upper-case letters and decimal digits on old machines like the Control Data Corporation (CDC) 1604, the CDC 6600, the Burroughs 5500, the IBM 7030 "Stretch" computer, and others. Show the differences, and explain why Unicode is an improvement.

26.5.3.(1) What object code is generated by these constants?

```
C1       DC    CU'ABC'
C2       DC    CUL12'ABC'
C3       DC    CUL2'ABC'
C4       DC    CUL8'A&&B''C'
```

[169] The Assembler actually used the 1148 EBCDIC code page for the EBCDIC-to-Unicode translation.

26.6. Unicode Instructions

We will describe three groups of instructions:

- String search, compare, and move instructions
- Translation instructions
- Format conversion instructions

26.6.1. String Search, Move, and Compare

The three instructions in Table 176 search, compare, and move, Unicode strings. While each has "Unicode" in its name, they actually just search for, compare, and move arbitrary two-byte values; there is no need for the operands to be Unicode characters! They are very similar to their single-byte counterparts: CLCLE, MVCLE, and SRST.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B2BE	SRSTU	RRE	Search String Unicode	EB8E	MVCLU	RSY	Move Long Unicode
EB8F	CLCLU	RSY	Compare Logical Long Unicode				

Table 176. Unicode string instructions

Their operand formats are:

```
SRSTU  R1,R2
MVCLU  R1,R3,D2(B2)
CLCLU  R1,R3,D2(B2)
```

Though each instruction manipulates two-byte operands, there is no requirement that they be halfword aligned.

26.6.1.1. Search String Unicode: SRSTU scans the second operand string addressed by register R_2, looking for a *pair* of bytes matching the rightmost two bytes of GR0 (the "test" character; the rest of GR0 must be zero). If a match is found, the R_1 register is set to its address. Because the second operand string can be very long, the CPU uses "Method B" to process part of the string before checking for interruptions.

To use SRSTU, put the test character in GR0, set the R_2 register to the address of the leftmost byte of the string to be scanned, and the R_1 register to the address of the first byte *after* the end of the string. The CPU uses this address to know when to stop the scan; otherwise, it could keep scanning byte pairs in memory until it found a match somewhere, or caused an unexpected interruption.

Table 177 gives the Condition Code settings after SRSTU:

CC	Meaning
1	Test character found; R_1 points to it
2	Test character not found before the byte addressed by R_1
3	Partial search with no match; R_1 unchanged, R_2 points to next byte to process

Table 177. CC settings for SRSTU instruction

On completion, either or both of the R_1 and R_2 registers may be updated:

- If the CC is 1, the R_1 register is updated and the R_2 register is unchanged.
- If the CC is 2, both the R_1 and R_2 registers are unchanged.
- If the CC is 3, R_1 is unchanged and R_2 is updated to the address of the next byte to be tested. You can then branch back to the SRST instruction to continue the search.

The operation of the SRSTU instruction is very similar to that of SRST, as sketched in Figure 235 on page 416, except that the second operand address is incremented by 2.

For example, suppose you want to scan the string at **MyData** to find the first occurrence of a Unicode “A” character:

```
UnicodeA Equ   X'0041'             Unicode capital letter A
         LAY   0,UnicodeA          Put test character in GR0
         LA    1,MyData            Set GR1 (R2) to string start
         LA    5,MyData+L'MyData   GR5 (R1) = byte past end
Repeat   SRSTU 5,1                 Scan the string
         JO    Repeat              Scan was incomplete, try again
         JH    NotFound            CC2, no match was found
         - - -                     CC1, GR5 now points to the A
```

If the Condition Code is 3, we simply branch back to the SRST to continue the search.

26.6.1.2. Move Long Unicode: Like MVCLE, the results of overlapping operands are unpredictable. The execution of MVCLU is the same as in Figure 229 on page 412, except that addresses are incremented by 2 and lengths are reduced by 2, and that two bytes are moved at each step. Also, the low-order 16 bits of the Effective Address are used as the pad “character”.

Normally both operand lengths are even, but MVCLU allows either operand to have odd length. The two-byte padding “character” is then described as the high-order padding byte followed by the low-order padding byte.

- If the operand lengths are equal but odd, MVCLU does the same as MVCLE.
- If the first (target) operand length is odd and is shorter than the third (source) operand, an odd number of bytes is moved.
- If the first operand is longer:
  - If its length is even, padding starts with the even padding byte.
  - If its length is odd, padding starts with the odd padding byte. This means that any additional padding is done with “proper” even-odd pairs of padding bytes.

MVCLU sets the Condition Code as shown in Table 178.

CC	Meaning
0	All bytes moved, operand lengths are equal
1	All bytes moved, operand 1 shorter; part of operand 2 was not moved
2	All bytes moved, operand 1 longer; operand 1 was padded
3	Some bytes moved; end of operand 1 not reached

Table 178. CC settings after MVCLU

For example, suppose we use MVCLU to initialize a large area of storage starting at **Work** to Unicode space characters:

```
         XR    0,0                 Source address in R3 will be ignored,
         XR    1,1                 ...because source length is zero
         LA    2,Work              Start of area to initialize in R1
         L     3,WorkSize          Length of work area in R1+1
Move     MVCLU 2,0,X'020'          Initialize with Unicode spaces
         JO    Move                Repeat until done
         - - -
WorkSize DC    A(BlockLen*NBlocks)     Length of work area
BlockLen Equ   32000               ...containing 32000 blocks,
NBlocks  Equ   20000               ...each 20000 bytes long
Work     DS    (NBlocks)CL(BlockLen)   Work area
```

Figure 247. Using MVCLU to initialize an area to Unicode spaces

26.6.1.3. Compare Logical Long Unicode: CLCLU operates in much the same way as CLCLE. However, operand lengths must be even, operand addresses are incremented by 2 and lengths are decremented by 2, and the 16-bit padding “character” is contained in the low-order two bytes of the Effective Address formed from the second operand.

CLCLU sets the Condition Code as shown in Table 179.

CC	Meaning
0	All bytes compared, operands equal, or both zero length
1	First operand low
2	First operand high
3	Some bytes compared without finding an inequality

Table 179. CC settings after CLCLU

To illustrate, we'll rewrite Figure 247 on page 442 to use CLCLU to test if the same field contains all Unicode spaces:

```
         XR    0,0              R3 address will be ignored,
         XR    1,1              ...because its length is zero
         LA    2,Work           Start of area to test in R1
         L     3,WorkSize       Length of test area in R1+1
Compare  CLCLU 2,0,X'020'       Compare entirely to pad bytes
         JO    Compare          Repeat until done
```

Figure 248. Using CLCLU to test for Unicode spaces

Warning!

Comparing and sorting character data in EBCDIC and Unicode can give very different results, because the character encodings are quite different.

26.6.2. Optional Operands (*)

Many Unicode-related instructions introduce *optional operands*, which have not been used in most of the instructions we've seen previously.

Unused fields in instructions are filled with zero bits by the assembler. But as System z has evolved, the CPU architects have sometimes needed to extend the function of an existing instruction. Rather than create a new instruction, some of these previously unused fields were assigned special bit-mask values.

It was important to avoid the problem that programs containing existing instructions now supporting new fields might need to be rewritten to handle the new operand specifications. To solve this, the new operand was made (a) optional and (b) the last operand of the assembler instruction statement. If the optional operand is omitted, the Assembler sets the optional field to zero, as usual. Thus, programs needing the enhanced function can specify the new operand, and existing programs not specifying the new operand continue to work without modification.

For example, an RRE-type instruction has the format shown in Table 180.

opcode		R_1	R_2

Table 180. RRE-type instruction

The extension of RRE-type instructions to support the new optional operand required a new instruction type: RRF, shown in Table 181 on page 444.

opcode	M_3		R_1	R_2

Table 181. RRF-format instruction with an optional operand

where M_3 is the optional operand; the notation used in the *z/Architecture Principles of Operation* describing the Assembler Language format of such instructions is illustrated in Figure 249, where the square brackets [ ] indicate that the operand is optional.

```
         mnemonic  R1,R2[,M3]
```

Figure 249. Assembler instruction statement for RRF-type instructions with an optional operand

If the optional operand is omitted, the Assembler fills the M_3 field of the instruction with zero bits, so that writing

```
         mnemonic  R1,R2
```

is the same as writing

```
         mnemonic  R1,R2,0
```

We will see examples of optional operands in some of the following Unicode instructions.

26.6.3. Translation

It is often useful to translate between Unicode and a single-byte character encoding like EBCDIC or ASCII; these instructions can help.

The four instructions for Unicode translation are listed in Table 182. Each has an optional operand in the form shown in Table 181 and Figure 249 above.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B993	TROO	RRF	Translate One to One	B992	TROT	RRF	Translate One to Two
B991	TRTO	RRF	Translate Two to One	B990	TRTT	RRF	Translate Two to Two

Table 182. Unicode translate instructions

These instructions are generalizations of TRE, which was described in "25.7. Translate Extended Instruction" on page 421, but there are interesting and significant differences:

- TRE tests a source-operand character, but these instructions test the function character from the translate table.
- The address of the translation table for TRE is specified in R_2, but is in GR1 for these four instructions. Unlike other translate instructions, you can specify different source and target operands, so the translation can be non-destructive.
- The source, target, and test characters can be either one or two bytes long. This means that the translation tables may be (much) larger:
  - TROO: Translate One to One (256-byte table)
  - TROT: Translate One to Two (512-byte table)
  - TRTO: Translate Two to One (64K-byte table)
  - TRTT: Translate Two to Two (128K-byte table)
- If the optional operand is 1, these instructions suppress the test for a source character matching the test character in GR0, and the translation is controlled only by the length of the first operand.
- TRE tests the character in GR0 *after* the result byte has been stored, but these four instructions test the function character against the character in GR0 *before* completing a character translation.
- If you are translating characters of the same length (one to one, or two to two), the source and target operands have the same length. Otherwise:

- If you translate One to Two, the target operand is twice as long as the source operand.
- If you translate Two to One, the target operand is half as long as the source operand.

- Unlike TR, TRT, and TRE, the translate tables must be aligned on a doubleword boundary.[170]
- If the target operand contains two-byte characters (for TROT and TRTT), the source character is shifted left internally by one bit before being added to the table address from GR1, to correctly address the two-byte translate table entries.

Some of these factors are summarized in Table 183.

Instruction	Source	Test Character	Table Entry	Table Size
TROO	1 byte	1 byte	1 byte	256
TROT	1 byte	1 byte	2 bytes	512
TRTO	2 bytes	2 bytes	1 byte	65,536
TRTT	2 bytes	2 bytes	2 bytes	131,072

Table 183. Arguments and translate tables for TRxx instructions

Table 184 summarizes how the registers are used by these four translate instructions:

Register	Contents
GR0	Test character, unless the optional operand is 1, in which case GR0 can be used for any other purpose.
GR1	Address of the translation table.
R_1	Address of the target operand where translated characters will be stored; must be even.
R_1+1	Length in bytes of the *second* (source) operand; must be an even number for TRTO, TRTT.
R_2	Address of the source operand to be translated.

Table 184. Registers used by TRxx instructions

All overlaps produce unpredictable results, whether of storage operands or of register assignments.

Table 185 describes the Condition Code settings after executing these instructions:

CC	Meaning
0	All characters translated; if the optional operand was 0 (test character comparison was performed), no function character matching the test character was found. R_1 points to the byte after the last target character, R_1+1 is zero, and R_2 points to the byte after the last source character,
1	If the optional operand was 0 (test character comparison was performed), a function character matching the test character was found. R_1+1 is decreased by the number of bytes processed *before* the character whose function character matched the test character; R_2 is incremented by the same number, and R_1 is increased by the number of bytes placed in the first operand.
3	Partial translation; branch back to the instruction to continue. Registers R_1, R_1+1, and R_2 have been adjusted so that translation will continue, eventually ending with CC 0 or 1.

Table 185. Condition Code settings for TRxx instructions

Remember: if the optional M_3 operand is 1, the test character is ignored.

[170] Originally, the translate tables for TRTO and TRTT had to be on a 4K-byte boundary, but this was quite inconvenient for most programs so the restriction was removed.

To illustrate, suppose TRTT is used to translate a string of DBCS characters to Unicode:

```
         LA    1,MapTbl          Point GR1 to the translation table
         LA    2,UString         Point GR2 to the Unicode result string
         LA    3,L'DBCSCon       Set GR3 to the second operand length
         LA    4,DBCSCon         Point GR4 to the DBCS source operand
TrDBCS   TRTT  2,4,1             Translate without a test character
         JO    TrDBCS            Repeat until all translations done
         - - -
DBCSCon  DC    G'<.A.B ... .Y.Z>' DBCS character string
UString  DS    CL(L'DBCSCon)     Unicode result has same length
         DS    0D                Doubleword alignment for table
MapTbl   DC    X'  ...  '        Mapping from DBCS to Unicode
```

Figure 250. Using TRTT to translate from DBCS to Unicode

Because the translate table at **MapTbl** could be up to 131,072 bytes long, we could take advantage of the fact that DBCS characters have representations between X'4040' and X'7FFE' to define a smaller table.

While these may seem very complex, the TROO instruction can be used in many places where TR would be inconvenient, because it lets you specify separate source and target operands.

To illustrate, suppose you must translate a long string of characters at **OldText** to another string at **NewText** while leaving the original string unchanged. Assume the string length is stored in the word at **TextLen** and that the required translation table starts at **TextTbl**. Then, Figure 251 shows instructions using MVC and TR compared to instructions using TROO.

```
* With MVC and TR                 |* With TROO
         L     3,TextLen          |          L     3,TextLen
         LA    2,NewText          |          LA    2,NewText
         LA    4,OldText          |          LA    4,OldText
         LA    1,255              |          LA    1,TextTbl
Repeat   CHI   3,256              |Repeat    TROO  2,4,1
         JNH   LastPart           |          JO    Repeat
         EX    1,MoveText         |Done      - - -
         EX    1,TRText           |
         AHI   2,256              |
         AHI   4,256              |
         AHI   3,-256             |
         J     Repeat             |
LastPart JNP   Done               |
         BCTR  3,0                |
         EX    3,MoveText         |
         EX    3,TRText           |
Done     - - -                    |
         - - -                    |
MoveText MVC   0(*-*,2),0(4)      |
TRText   TR    0(*-*,2),TextTbl   |
```

Figure 251. Translating a long string with TR and MVC, and with TROO

With TROO, the CPU adjusts the operand registers automatically, while with MVC and TR you must write instructions to do the updates.

26.6.4. Conversion Among Transformation Formats (*)

As mentioned in Section 26.5, the Unicode standard defines 8-, 16-, and 32-bit encoding formats; each is useful in different contexts. One problem in transmitting Unicode data across networks is that some of the byte codes used for the 16-bit or 32-bit Unicode characters also have meaning as network control codes. It may be necessary to transform a UTF-16 or UTF-32 encoding to UTF-8 for transmission over a network, and the receiver can transform the byte stream back to UTF-16 or UTF-32 (or even work directly with the received UTF-8 byte stream).

The bits of a UTF-16 character are sometimes represented as shown in Figure 252:

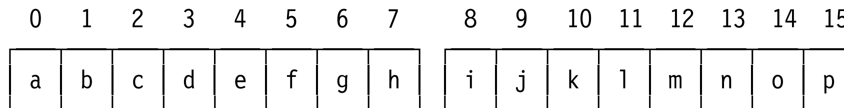


Figure 252. Bits of a UTF-16 Unicode character

To see why UTF-8 is needed, consider the Cyrillic character "P" with representation `U+0420`. If this was transmitted as single byte data, the single `X'04'` byte could be interpreted as an End of Transmission (EOT) flag! Thus, the UTF-8 encoding transforms a UTF-16 character to one, two, three, or four bytes as follows:

- If the UTF-16 character has the form `B'00000000 0jklmnop'` (it lies between `U+0000` and `U+007F`), it is transformed into a single byte `B'0jklmnop'`.
- If the UTF-16 character has the form `B'00000fgh ijklmnop'` (it lies between `U+0080` and `U+07FF`), it is transformed into the two bytes `B'110fghij 10klmnop'`.
- If a UTF-16 character lies between `U+0800` and `U+D7FF`, or lies between `U+DC00` and `U+FFFF`, it is transformed into the three bytes `B'1110abcd 10efghij 10klmnop'`.
- Because the 16 UTF-16 bits were not sufficient to encode all required characters, forms with scalar values greater than or equal to `U+10000` were added; they are represented by surrogate pairs, as illustrated in Figure 253. The four bytes of surrogate pairs have these bit patterns:

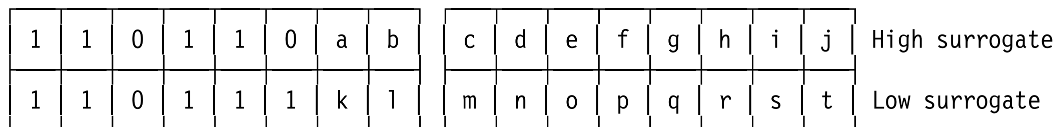


Figure 253. Bits of a UTF-16 Unicode surrogate pair

Given a high surrogate that lies between `U+D800` and `U+DBFF` and a low surrogate that lies between `U+DC00` and `U+DFFF`, the pair is transformed into the four UTF-8 bytes `B'11110uvw 10xyefgh 10ijklmn 10opqrst'`, where `uvwxy = abcd+1`.

While this transformation is complicated, it guarantees that no UTF-16 character will be mistaken for a control code, and that the receiver can immediately find where a character encoded in UTF-8 form begins. The initial bits of the first UTF-8 byte indicates how many bytes follow it in a multi-byte sequence.

The Unicode and ISO standards also define a UTF-32 standard that uses 32 bits for each character, and does not need surrogates. The UTF-32 representation of a "normal" UTF-16 character (as shown in Figure 252) with encodings from `U+0000` to `U+DBFF` and from `DC00` to `U+FFFF`, simply appends two high-order bytes of zeros to the UTF-16 character.

For UTF-16 surrogate pairs, the mapping is more complex: the four bytes shown in Figure 253 are mapped into the form shown in Figure 254 on page 448, where the bits named `uvwxy` have value `abcd+1`.

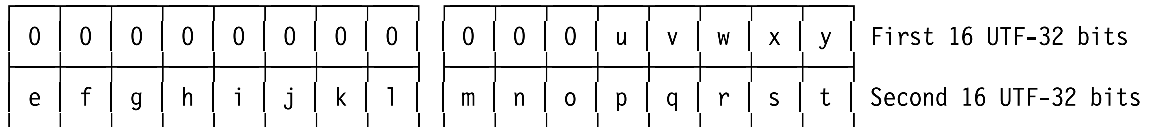


Figure 254. Bits of a UTF-32 Unicode character from a UTF-16 surrogate pair

The System z instructions that convert among the three UTF encodings are shown in Table 186. Two of the instructions have two names, because CUUTF and CUTFU were implemented before the other four; the original two were then renamed so that all six instruction names are consistent. The old names are retained for compatibility.[171]

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B2A7	CU12, CUTFU	RRF	Convert UTF-8 to UTF-16	B9B0	CU14	RRF	Convert UTF-8 to UTF-32
B2A6	CU21, CUUTF	RRF	Convert UTF-16 to UTF-8	B9B1	CU24	RRF	Convert UTF-16 to UTF-32
B9B2	CU41	RRF	Convert UTF-32 to UTF-8	B9B3	CU42	RRF	Convert UTF-32 to UTF-16

Table 186. Unicode format conversion instructions

The CU12, CU14, CU21, and CU24 instructions have an optional M_3 operand; CU41 and CU42 do not.

The R_1 and R_2 registers must be even, and contain the addresses of the first and second operands; the next higher-numbered registers R_1+1 and R_2+1 contain the operand lengths.

The CU41 and CU42 instructions have the form shown in Table 180 on page 443, and the CU12, CU14, CU21, and CU24 instructions have the form shown in Table 181 on page 444.

When these instructions were first implemented, the Unicode Standard did not forbid invalid forms. Further revisions of the Standard made those forms invalid, so an optional operand was added to CU12, CU14, CU21, and CU24 to let you choose whether or not "well-formedness" should be enforced. If the optional operand is one, checking is done for well-formedness.[172] Further details may be found in the *z/Architecture Principles of Operation*.

Table 187 shows the Condition Code settings after executing these instructions:

CC	Meaning
0	Entire second operand was processed.
1	End of the first operand was reached.
2	An invalid UTF-8 character was found; or an invalid low surrogate was found; or an invalid UTF-32 character was found.
3	Operation incomplete; branch back to it to complete the operation.

Table 187. CC settings after Unicode format conversion instructions

Exercises

26.6.1.(2) What do you think will happen if the B_2 register of a CLCLU or MVCLU instruction is the same as R_1 or R_3 or R_1+1 or R_3+1?

26.6.2.(1) Can both operands of CLCLU be padded?

[171] Originally, only the two instructions CUUTF and CUTFU were defined, meaning "Convert Unicode to UTF" and "Convert UTF to Unicode", respectively. At that time, "UTF" meant UTF-8, and "Unicode" meant UTF-16.

[172] A well-formed multi-byte UTF-8 character requires that each byte after the first start with B'10'.

26.6.3.(2)+ Write a sequence of instructions to find the last nonblank UTF-16 Unicode character in a string of UTF-16 Unicode characters stored at **UData**. If no nonblank characters are found, branch to **AllBlank**.

26.6.4.(1)+ Write instructions to copy a string of Unicode characters from **UHere** to **UThere**.

26.6.5.(1) What UTF-16 Unicode characters can be used for padding if you use a CLCLU or MVCLU instruction with B_2=0?

26.6.6.(3) In Figure 250 on page 446, what changes to the first LA instruction would be needed if the translate table at **MapTbl** starts with the DBCS character corresponding to the DBCS character with representation X'4040'?

26.6.7.(2)+ Suppose one of the four TRxx instructions is used with the optional operand set to 1. Can GR0 be used for the R_2 operand?

26.6.8.(2) Can TRTT be used to simulate TROO? Explain why or why not.

26.6.9.(2)+ Suppose you are given a string of bytes in UTF-8 format, and you must start at some arbitrary position in the string. How can you locate the starting byte of the nearest valid UTF-8 character? What is the maximum number of bytes you could skip before finding that valid starting byte?

26.6.10.(3) It is claimed that a binary sort of UTF-8 character strings gives the same ordering as a binary sort of UTF-16 scalar values, so long as there are no surrogates. Create examples to show that this is or is not true.

26.6.11.(2) GR5 contains a halfword value that is to be sought in a table of halfwords starting at **HWList**. Write a sequence of instructions using SRSTU to locate the entry in the table that matches the halfword in GR5.

26.6.12.(4) Create a translation table for mapping "Assembler EBCDIC" code page 037 to UTF-16. Which instruction would you use?

26.6.13.(5) Many programs convert strings of bytes to pairs of EBCDIC characters representing the hexadecimal digits of each byte. (Exercise 15.6.5 is a typical example.) Using a single DC statement, create a translate table to be used by a TROT instruction like

```
         LA    1,PH               Address of 1-to-2 translation table
         LA    2,Target           Address of target string
         LA    3,L'Source         Length of source string
         LA    4,Source           Address of source string
Repeat   TROT  2,4,1              Translate each byte to two
         JO    Repeat             Repeat if incomplete
```

that converts source bytes to their representative pairs of EBCDIC characters. For example, the table should start with the characters C'00010203' (that is, X'F0F0F0F1F0F2F0F3'), and end with the characters C'FCFDFEFF' (that is, X'C6C3C6C4C6C5C6C6').

26.6.14.(2) Suppose the CU24 and CU42 instructions did not exist. How could you translate between UTF-16 and UTF-32?

26.6.15.(2) Show how the bit patterns of the first UTF-8 byte can be used to determine how many following bytes are part of the same Unicode character.

26.6.16.(2) Suppose you have a large table of 8-bit signed binary integers in a string of bytes starting at **B**, whose length is defined by the EQUated symbol **NB**. Create a translation table starting at **TBH** that can be used by a TROT instruction to convert the signed bytes into a table of **NB** signed halfword binary integers starting at **H** having the same values.

26.7. Translate and Test Extended

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B2BF	TRTE	RRF	Translate and Test Extended	B2BD	TRTRE	RRF	Translate and Test Reverse Extended

Table 188. Translate and Test Extended instructions

Both instructions have the same operand format.

```
Mnemonic  R1,R2[,M3]
```

The operand assignments are:

R_1 The address of the first character of the string of characters to be tested. (The specific meaning of "character" in this context will be described below.) The R_1 register must be even.

R_1+1 The length of the first operand in bytes (not characters!).

R_2 When a first-operand character matches a nonzero function code from the function-code ("translate") table, that code is placed in the R_2 register.

GR1 The address of the function-code table.

> **Warning!**
>
> The table must be on a doubleword boundary, because the instructions ignore the three-low-order bytes of this address. If your table isn't doubleword aligned, your results will likely be very incorrect.

M_3 The optional 4-bit M_3 operand assigns names A, F, and L to the first three bits; the rightmost bit should be zero, as shown. If M_3 is omitted, the M_3 field is set to zero.

A	F	L	///

They have these meanings:

A 0: Argument characters are 1 byte.

1: Argument characters are 2 bytes, and the argument string length in R_1+1 must be even.

F 0: Function codes are 1 byte.

1: Function codes are 2 bytes.

L 0: The full range of argument and function codes allowed; the range depends on the lengths of the argument and the function codes.

1: If an argument value is greater than 255, the function code is assumed to be zero.

The various combinations of the A, F, and L bits mean that the function-code table will have different sizes, as shown in Table 189. (A table entry "—" means that the bit can have value 0 or 1 without affecting the table size.)

A	F	L	Table size (bytes)
0	0	—	256
0	1	—	512
1	0	0	65536
1	1	0	131072
1	0	1	256
1	1	1	512

Table 189. Function-code table sizes for TRTE, TRTRE

In the two cases where L=1, the fact that A=1 means that argument characters are two bytes long, potentially meaning that they could correspond to function codes that are also two bytes long if F=1 also. Because L=1, any argument value greater than 255 (that is, with a nonzero value in the left byte) is ignored, and the instructions treat the function code as being zero.

For TRTE, the argument string is scanned from left to right, so the R_1 address is simply set to the address of the first byte in the string. Depending on the A bit, arguments are scanned one byte at a time (A=0) or two bytes at a time (A=1).

For TRTRE, the argument string is scanned from right to left. If the A bit is zero, the R_1 address points to the rightmost byte of the argument string (the last “character”). If however the A bit is one, meaning that argument characters are two bytes long, the R_1 address points to the *next to last byte* of the string: that is, to the last *character* of the string.

The Condition Code settings for TRTE and TRTRE are shown in Table 190.

CC	Meaning
0	All first operand bytes processed; all function bytes were zero
1	A nonzero function byte was found
3	First operand partially processed; try again

Table 190. Condition code settings for TRTE, TRTRE

A simple example of TRTE is shown in Figure 255. Assume you must scan an integer expression encoded in Unicode UTF-16 characters, with operators and parentheses. Because L=1, we need only 256 table entries; and all the Unicode characters being sought have values less than X'0040'.

```
A        Equ   1                  (Argument character length) -1
F        Equ   1                  (Function code length) +1
L        Equ   1                  Ignore argument characters >255
M3       Equ   8*A+4*F+2*L        M3 mask field value

         LAY   4,String           Address of characters to scan
         LHI   5,L'String         Length to scan
         XR    6,6                For inserted function code
         LAY   1,Tbl              Address of function code table
Scan     TRTE  4,6,M3             Scan the string
         JO    Scan               Repeat of nothing found yet
         LAY   0,Process          Address of start of process code
         ALR   6,0                Form branch address
         BR    6                  Branch to select processing routine
         - - -
Tbl      DC    0D,(X'28')AL2(0)   Ignore uninteresting characters
         DC    AL2(LP-B)          Left parenthesis
         DC    AL2(RP-B)          Right parenthesis
         DC    AL2(M-B)           * (Multiplication)
         DC    AL2(P-B)           + (Addition)
         DC    AL2(0)             Ignored character
         DC    AL2(S-B)           - (Subtraction)
         DC    AL2(0)             Ignored character
         DC    AL2(D-B)           / (Division)
         DC    10AL2(N-B)         Numeric digit
         DC    (X'FF'-X'39')AL2(0)   Ignored characters
```

Figure 255 (Part 1 of 2). Example of using TRTE

```
Process  DC   0F                 Branch to process routines
B        B    Nothing            Nothing interesting found
P        B    Plus               + operator
S        B    Minus              - operator
N        B    Numeric            Numeric digit
M        B    Mult               * operator
RP       B    RParen             Right parenthesis
D        B    Div                / operator
LP       B    LParen             Left parenthesis
*                                Processing routines follow...
Nothing  DC   0H ...             Processing in case nothing found
Numeric  DC   0H ...             Processing for digit
Plus     DC   0H ...             Processing for +
Minus    DC   0H ...             Processing for -
Mult     DC   0H ...             Processing for *
Div      DC   0H ...             Processing for /
RParen   DC   0H ...             Processing for )
LParen   DC   0H ...             Processing for (
String   DS   XL4096             String to be scanned
```

Figure 255 (Part 2 of 2). Example of using TRTE

The technique of using offsets to the processing routines as function-code values allows you to arrange the table of branch instructions in any order you like (except that the "not found" case must be first).

Exercises

26.7.1.(1)+ What size should the function-code table be if you are translating 2-byte characters, and have set `L = 1`? Why?

26.7.2.(4) Suppose the three symbols A, F, and L in the M_3 mask have been defined as absolute symbols with values 0 or 1. Suppose also that you must reserve space for a function-code table with size depending on the values of A, F, and L, as illustrated in Table 189 on page 450. Write an EQU statement defining a symbol **TL** that will contain the length of the reserved area for the table.

26.7.3.(3) Compare TRT and TRTE with respect to these characteristics (making a table may help).

1. Operand length range
2. How operand length is specified
3. How the first operand is addressed
4. How the function-code table is addressed
5. Where a nonzero function code is placed, and what happens to the rest of that register
6. Where the address of the corresponding argument character is placed
7. The length and alignment of the function-code table
8. Condition Code settings and the conditions they represent

26.7.4.(3) Repeat Exercise 26.7.3 for TRTR and TRTRE.

26.7.5.(2) Consider the possible combinations of argument characters and function codes for TRTE in Table 189 on page 450. Which of those combinations are related to the four instructions TROO, TROT, TRTO, and TRTT? In what ways are they related and not related?

26.7.6.(3) Modify the function-code table definion in Figure 255 on page 451 so that the size of each entry in the table depends only on the value of the F bit. That is, you need only change the EQU statement defining F and reassemble.

26.8. Byte Reversal and Workstation Data

All our previous examples using binary data have assumed that the most significant bits are found in the byte with the lowest address, with significance decreasing at higher addresses. Some other processor architectures store numeric data in the opposite direction: the byte with the least significant bits are at the lowest address, and significance *increases* in the bytes at higher addresses.

The significance of bits within a byte is the same in either case: the high-order bit has the greatest significance.[173]

The choice of byte order for binary data is sometimes called the "Endian" question.[174] Because System z stores the most significant bits at the lowest address, it is called a "Big-Endian" processor (i.e., "big end first"). Many early microprocessors could handle only 4 bits or a single byte at a time, and because bits are added starting with the least significant, it was more economical to address the (numerically) lowest-order byte (i.e., "little end first"). Many personal computers and workstations are "Little-Endian".

For example, if a word containing `X'87654321'` is stored in a Big-Endian processor memory starting at address **A**, we would see the bytes in storage as in Figure 256:

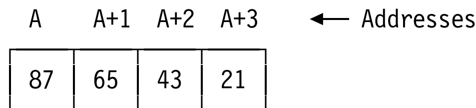


Figure 256. Big-Endian storage representation of X'87654321'

However, on a processor storing data with the least significant bits at the lowest address (a "Little-Endian" processor) we would see the bytes in storage as in Figure 257:

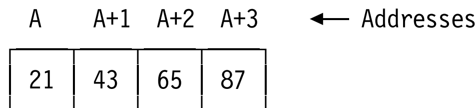


Figure 257. Little-Endian storage representation of X'87654321'

When your program manipulates multi-byte data, you'll want to know where it originated.

26.8.1. Byte-Reversing Instructions

The byte-reversing instructions are listed in Table 191. They are provided because System z processors must often exchange data with workstations and personal computers that store some types of data in byte-reversed order.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E31E	LRV	RXY	Load Reversed (32)	B91F	LRVR	RRE	Load Register Reversed (32)
E30F	LRVG	RXY	Load Reversed (64)	B90F	LRVGR	RRE	Load Register Reversed (64)
E31F	LRVH	RXY	Load Halfword Reversed (16)	E33F	STRVH	RXY	Store Halfword Reversed (16)
E33E	STRV	RXY	Store Reversed (32)	E32F	STRVG	RXY	Store Reversed (64)

Table 191. Byte-reversing load and store instructions

[173] But be careful: data on some processors sometimes numbers bits from right to left, rather than the System z convention from left to right. Right-to-left numbering has the advantage that a bit's number is the same as the power of 2 it represents.

[174] The term "Endian" was taken from Jonathan Swift's *Gulliver's Travels*, where the kingdoms of Lilliput and Blefuscu were permanently at war over the correct way to eat a boiled egg. The Lilliputian "Little-Endians" insisted on opening the egg at the sharp end, and the Blefuscudian "Big-Endians" insisted on the rounded end.

They behave like normal Load and Store instructions, except that the left-to-right order of the bytes is reversed. This is illustrated in Figure 258 on page 454.

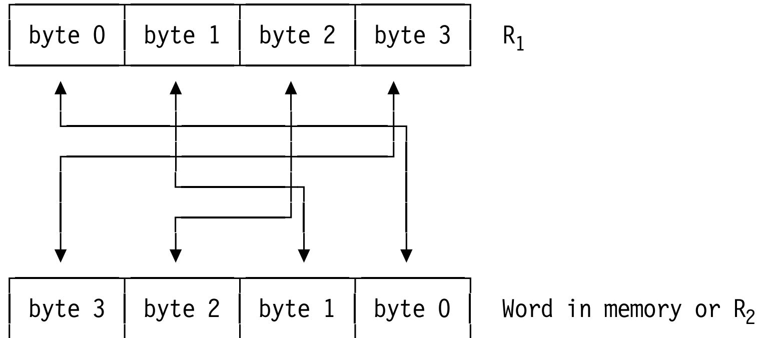


Figure 258. Byte reversal by LRV, LRVR, and STRV instructions

These instructions convert two, four, or eight bytes between “Big-Endian” and “Little-Endian” format quickly and efficiently for processing. For example, suppose the word in Figure 256 on page 453 is stored at **A**. Then, executing

```
         LRV   0,A                  c(GR0) = X'21436587'
```

will load GR0 with the pattern shown in Figure 257 on page 453. Byte reversal can also be done when storing the contents of a general register into memory:

```
         L     0,=X'12345678'       c(GR0) = X'12345678'
         STRV  0,Rev                c(Rev) = X'78563412'
```

The LRVR instruction is similar to LRV, except that the second operand comes from GR R_2 rather than from memory.

The LRVH and STRVH instructions are similar to LRV and STRV, except that only the two rightmost bytes of GR R_1 are involved. This is illustrated in Figure 259.

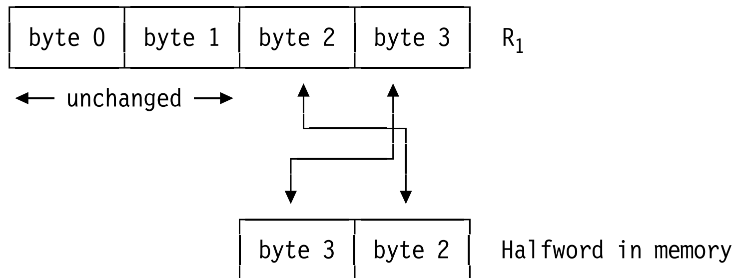


Figure 259. Byte reversal by LRVH and STRVH instructions

For example:

```
         LRVH  1,=X'1234'           c(GR1) = X'xxxx3412'
```

where xxxx is the original data in the two high-order bytes of GR1.

LRVH is unusual in this respect: unlike the other Load Halfword instructions, it does *not* propagate the sign bit of the leftmost bit of the (reversed) halfword just loaded. Thus, it might better be thought of as the “Insert Halfword Reverse” instruction, because the rest of the R_1 register is unchanged.

The 64-bit instructions LRVG, LRVGR, and STRVG are the 64-bit equivalents respectively to LRV, LRVR, and STRV: the operands are 8 bytes long rather than 4. Otherwise, their operation is the same.

Processing halfword, word, and doubleword data with these instructions is straightforward.

If the fields in a data item do not align neatly on byte boundaries, processing "Endian" data can be much more difficult. Suppose a word in memory containing four different binary integers has the format shown (in Big-Endian format) in Figure 115 on page 249. As before, the bits are lettered, but now we must number the individual bits of the integers A, B, C, and D, as in Figure 260 on page 455. The 9 bits of A are numbered A0-A8; the 4 bits of B are numbered B0-B3; the 13 bits of C are numbered C0-Cc; and the 6 bits of D are numbered D0-D5.

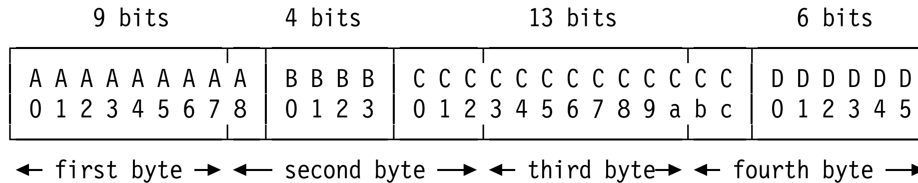


Figure 260. Four integers packed in a Big-Endian 32-bit word

Thus, the first byte contains bits A0-A7; the second byte contains bits A8, B0-B3, and C0-C2; the third byte contains bits C3-Ca; and the fourth byte contains bits Cb-Cc and D0-D5.

Suppose these four bytes are reversed and sent to a Little-Endian processor; the data would then look like this:

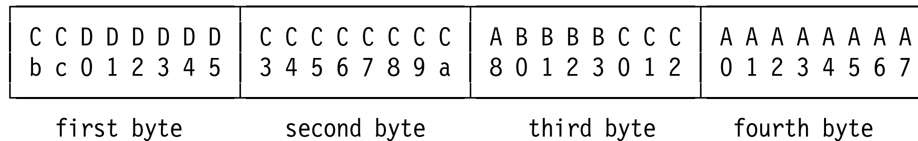


Figure 261. The same four integers packed in a Little-Endian 32-bit word

You can imagine the difficulties a program on a Little-Endian processor might have in extracting the four integer values. Conversely, if your program receives a Little-Endian word in which overlapping bit fields are specified as in Figure 260, your System z program would see the word in Figure 261.

Something to Check

Before processing data, be sure you know its "Endianness", as well as its character representation.

Exercises

26.8.1.(2) Suppose the two rightmost bytes of GR2 contain a 2-byte Little-Endian integer. What will these two instructions do?

```
LRVR  2,2
SRA   2,16
```

What will happen if the SRA instruction is replaced by SRL?

26.8.2.(3) Suppose the data shown in Figure 260 is actually in Little-Endian format, so you find it on your System z processor in the word at **OddData** in the form shown in Figure 261. Write a sequence of instructions to extract the four (unsigned) integers, and store them in halfwords named **A**, **B**, **C**, and **D**.

26.8.3.(1) Put the 4 bytes of the word at DPG into GR1 in reverse order. (See Exercise 17.2.5.)

26.8.4.(2)+ An 80-byte character string is stored at **InRec**. Write a loop using byte-reversing instructions (not using MVCIN) to reverse the bytes and store them at **RevRec**.

26.8.5.(3) Suppose you have a string of bytes starting at **ByteStr** whose length, stored at **SLen**, is known to be a multiple of two. Write a loop using byte-reversing instructions (as in Exercise 26.8.4) to store the reversed bytes at **RevStr** executing as few instructions as possible. For

example, you might reverse as many groups of 8 bytes as possible using the 8-byte instructions, then handle the remainder as efficiently as you can.

26.9. Summary

The extended instructions for Unicode are summarized in Table 192.

Function	Instruction	Stop Conditions
Move	MVCLU	End of first operand
Compare	CLSTU	End of longer operand, or unequal comparison
Search	SRSTU	End of second operand, or stop character found

Table 192. Extended instructions for Unicode data

The Unicode-based translation instructions are summarized in Table 193.

Function	Operand 1	1 byte		2 bytes	
	Operand 2	1 byte	2 bytes	1 byte	2 bytes
Translate		TROO	TRTO	TROT	TRTT
Translate and Test Extended		TRTE		TRTE	
Translate and Test Reverse Extended		TRTRE		TRTRE	

Table 193. Unicode-based translate instructions

The Unicode format conversion instructions are summarized in Table 194.

Function	Operand 1	1 byte		2 bytes		4 bytes	
	Operand 2	2 bytes	4 bytes	1 byte	4 bytes	1 byte	2 bytes
Convert Format		CU21 CUUTF	CU41	CU12 CUTFU	CU42	CU14	CU24

Table 194. Unicode format conversion instructions

The byte-reversing instructions are summarized in Table 195.

Function	Operand1	4 bytes		8 bytes
	Operand2	2 bytes	4 bytes	8 bytes
Load (from memory)		LRVH	LRV	LRVG
Load (from register)			LRVR	LRVGR
Store		STRVH	STRV	STRVG

Table 195. Summary of byte-reversing instructions

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
CLCLU	EB8F
CU12, CUTFU	B2A7
CU14	B9B0
CU21, CUUTF	B2A6
CU24	B9B1
CU41	B9B2
CU42	B9B3

Mnemonic	Opcode
LRV	E31E
LRVG	E30F
LRVGR	B90F
LRVH	E31F
LRVR	B91F
MVCLU	EB8E
SRSTU	B2BE
STRV	E33E

Mnemonic	Opcode
STRVG	E32F
STRVH	E33F
TROO	B993
TROT	B992
TRTE	B9BF
TRTO	B991
TRTRE	B9BD
TRTT	B990

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
B2A6	CU21, CUUTF
B2A7	CU12, CUTFU
B2BE	SRSTU
B90F	LRVGR
B91F	LRVR
B990	TRTT
B991	TRTO

Opcode	Mnemonic
B992	TROT
B993	TROO
B9B0	CU14
B9B1	CU24
B9B2	CU41
B9B3	CU42
B9BD	TRTRE
B9BF	TRTE

Opcode	Mnemonic
E30F	LRVG
E31E	LRV
E31F	LRVH
E32F	STRVG
E33E	STRV
E33F	STRVH
EB8E	MVCLU
EB8F	CLCLU

Terms and Definitions

ASCII
American Standard Code for Information Interchange, an 8-bit encoding.

BCD
Binary Coded Decimal. (1) A 4-bit encoding of the decimal digits 0-9 used in packed decimal arithmetic. (2) A 6-bit character encoding used on many data processing systems prior to System/360's introduction of an 8-bit byte with EBCDIC encoding.

Big-Endian
A representation of numbers in which the value of the digits at successively higher addresses have *lower* significance; the digits have decreasing significance from left to right. The representation used on System z.

Code Page
A defined encoding of characters and control codes.

DBCS
See "Double-Byte Character Set".

Double-Byte Character Set
A 16-byte EBCDIC encoding of a character set having many more characters than can be accommodated in 8 bits.

EBCDIC
Extended Binary Coded Decimal Interchange Code, an 8-bit encoding of characters and control codes.

Glyph
The printed or displayed form of a character that can be formed with various properties. For example, the glyphs A, *A*, **A**, ***A***, A are representations of the character “A”, in upper case forms (normal, italic, bold, bold italic, and “small caps”).

Little-Endian
A representation of numbers in which the value of the digits at successively higher addresses have *greater* significance; the digits have increasing significance from left to right.

SBCS
See “Single-Byte Character Set”.

Shift-In
An `X'0F'` byte code in a stream of DBCS byte pairs indicating that the following single bytes are SBCS-encoded.

Shift-Out
An `X'0E'` byte code in a stream of SBCS bytes indicating that the following pairs of bytes are DBCS-encoded.

Single-Byte Character Set
An 8-bit encoding of a character set.

Syntactic Character Set
A set of 82 characters with the same encodings across all EBCDIC Code Pages.

Unicode
An international standard encoding of (almost) all characters, represented as groups of 8-bit bytes (UTF-8), one or two byte pairs (UTF-16), or 32-bit units (UTF-32).

Chapter VIII: Zoned and Packed Decimal Data and Operations

```
VV        VV  IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
VV        VV  IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
VV        VV      II          II          II
VV        VV      II          II          II
VV        VV      II          II          II
VV        VV      II          II          II
VV        VV      II          II          II
 VV      VV       II          II          II
  VV    VV        II          II          II
   VV  VV         II          II          II
    VVVV      IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
     VV       IIIIIIIIII  IIIIIIIIII  IIIIIIIIII
```

The four sections of this chapter discuss the zoned and packed decimal number representations and typical operations on each.

- Section 27 discusses the zoned and packed decimal representations and instructions that convert between them.
- Section 28 gives an overview of the principles of packed decimal arithmetic, to help you understand the operation of the instructions in Sections 29 and 30.
- Section 29 describes the instructions that perform packed decimal arithmetic operations.
- Section 30 describes instructions used to convert data between packed decimal and binary, and between packed decimal and character strings.

The other System z decimal data format, decimal floating-point, will be discussed in the next chapter.

27. Zoned and Packed Decimal Representations

```
 2222222222   777777777777
222222222222  777777777777
22        22  77        77
          22            77
          22           77
         22            77
       22             77
     22               77
   22                77
 22                  77
222222222222        77
222222222222        77
```

In this section we examine the *zoned decimal* and *packed decimal* representations of data, which are useful in applications requiring decimal arithmetic, compactness, selectable precision, and simplicity. (Packed decimal is different from decimal floating-point, which is discussed in the next chapter.) We'll start with the instructions in Table 196.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
D1	MVN	SS	Move Numerics	D3	MVZ	SS	Move Zones
F2	PACK	SS	Pack	F3	UNPK	SS	Unpack
E9	PKA	SS	Pack ASCII	EA	UNPKA	SS	Unpack ASCII
E1	PKU	SS	Pack Unicode	E2	UNPKU	SS	Unpack Unicode

Table 196. Basic packed and zoned decimal instructions

The zoned and packed decimal data representations of System z provide simple and economical ways to store decimal data that will not take on a large range of values. There are no instructions that perform arithmetic with zoned decimal data; its main use is as an intermediate step between the internal packed decimal representation (with which arithmetic is possible), and an external representation such as character data.

27.1. Zoned Decimal Representation

First, two definitions. The two hexadecimal digits of a byte are known as the "zone" (high-order) digit and the "numeric" (low-order) digit, represented by Z and n in Figure 262.

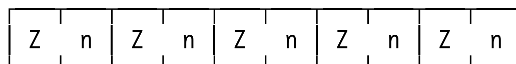


Figure 262. Zone and numeric digits of a byte

The MVN and MVZ instructions are almost identical to MVC: rather than moving entire bytes, they move only the numeric digits or only the zone digits, respectively. To illustrate, suppose we define a string of bytes at the constant named **RandomF** and move the numeric and zone digits to byte strings named **Numerics** and **Zones** respectively.

```
         MVN   Numerics,RandomF   Move numeric digits
         MVZ   Zones,RandomF      Move zone digits
         - - -
RandomF  DC    XL8'FEDCBA987654321F'     Source operand
Numerics DC    XL8'0'             Numerics
Zones    DC    XL8'0'             Zones
```

Figure 263. Example of MVN and MVZ instructions

Then, the contents of the two byte strings will be

```
c(Numerics) = X'0E0C0A080604020F'
c(Zones)    = X'F0D0B09070503010'
```

The two following examples illustrate some simple uses.

1. Convert the non-negative halfword integer at N to a string of five EBCDIC characters beginning at **NDec**, which give the decimal representation of the contents of the halfword at **N**.

```
XXX      LH    1,N                Number to be converted in odd reg
         LA    2,L'NDec           Use GR2 to count number of digits
XXX      SR    0,0                Clear high-order (even) register
         D     0,=F'10'           Generate a digit
         STC   0,NDec-1(2)        Store digit in output string
         BCT   2,XXX              Count and branch until done
         MVZ   NDec,=(L'NDec)X'FF'    Attach zones for EBCDIC
         - - -
NDec     DS    CL5                Converted result
N        DC    H'12345'           Number to convert
```

   We could have used literals such as =5C'0' or =5C'9' in the MVZ instruction, with the same results, as would any literal with X'F' in the zone digit.

2. Convert the five-digit decimal number in EBCDIC form at **NDec** to a fullword binary integer, and store it at **MM**.

```
RW       EQU   0                  Work register for inserting digits
RNum     EQU   1                  Value accumulated in GR1
RC       EQU   2                  Count digits in GR2
RDP      EQU   3                  Digit pointer
         MVN   Temp,NDec          Move numeric portions of digits
         LA    RDP,Temp           Address of current digit in RDP
         LA    RC,L'NDec          Number of digits to be processed
         SR    RW,RW              Clear RW for digits
         SR    RNum,RNum          And RNum for number being generated
Mult     MH    RNum,=H'10'        Multiply accumulated part by 10
         IC    RW,0(,RDP)         Insert digit from input, unzoned
         AR    RNum,RW            Add to partial sum
         LA    RDP,1(,RDP)        Increment digit address
         BCT   RC,Mult            Count and loop
         ST    RNum,MM            Store result
         - - -
NDec     DC    C'12345'           Value to be converted
Temp     DC    XL(L'NDec)'0'      Zones pre-zeroed, digits moved in
MM       DS    F                  Binary result
```

   As we will see, the CVD and CVB instructions considerably simplify the conversion of numbers between binary and decimal formats.

These instructions are sometimes used in processing packed decimal data.

Calling the left and right hex digits of a byte the “zone” and “numeric” digits might seem to limit the use of MVZ and MVN to packed and zoned decimal data. However, they simply move the left or right digits of a string of bytes, no matter what data type they represent.

The zoned decimal representation actually differs little from the familiar EBCDIC representation for characters. For example, the *EBCDIC* representation of the decimal characters 12345 is

F 1	F 2	F 3	F 4	F 5

The *only* difference between the zoned decimal and EBCDIC representations of the digits 12345 is in the treatment of the zone digit of the *rightmost* byte. In the EBCDIC representation, the zone digit is X'F', as illustrated above. In the zoned decimal representation, however, this digit may be any of the six hexadecimal digits A, B, C, D, E, or F; they are treated as the *sign* of the zoned number, as indicated in Figure 264.

Sign Digit	Sign
A	+
B	–
C	+
D	–
E	+
F	+

Figure 264. Zoned decimal sign conventions

Of the six possible sign digits, the preferred values are X'C' for "+" and X'D' for "–". Thus, the *zoned* decimal representation of +12345 is

F 1	F 2	F 3	F 4	C 5

Table 197 contains some some examples of zoned decimal constants. As these examples illustrate, leading zeros may appear in a zoned decimal number without affecting its value.

Value	Representation
+12345	X'F1F2F3F4C5'
-12345	X'F1F2F3F4D5'
+003	X'F0F0C3'
-0999	X'F0F9F9D9'

Table 197. Examples of zoned decimal data

If these byte strings are printed, the last character in each will be a letter!

Figure 265 gives a pictorial representation of a zoned decimal number, where "Z" represents a zone digit, "S" represents a sign digit, and "d" represents a numeric (decimal) digit.

Z d	Z d	Z d	Z d	S d

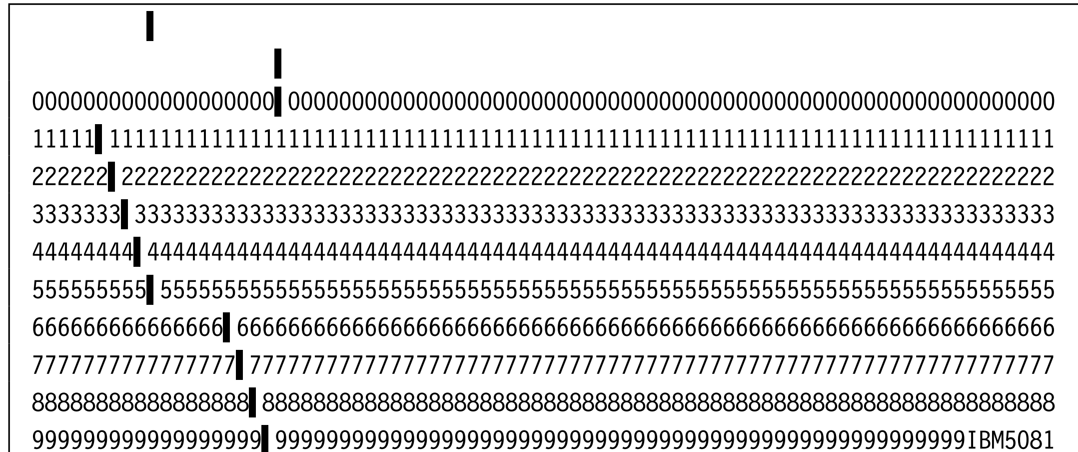
Figure 265. A zoned decimal number

All of the decimal characters we saw in previous sections used X'F' for their zone digits Z. Other zone digits are used for ASCII and Unicode numeric character data.

27.1.1. Why Zoned Decimal Is The Way It Is (*)

The reason for the choice of preferred zones comes from the days of punched cards. To save space in the card column containing the last character of a numeric field, by convention a hole corresponding to the desired digit and another *zone punch* appeared in one of the two top rows of the same card column.

For example, a punched card containing two 10-character fields containing the right-adjusted numbers +12345 and −67890 would appear as shown in Table 198:



```
         █
                   █
0000000000000000000█000000000000000000000000000000000000000000000000000000000000
11111█11111111111111111111111111111111111111111111111111111111111111111111111111
222222█2222222222222222222222222222222222222222222222222222222222222222222222222
3333333█333333333333333333333333333333333333333333333333333333333333333333333333
44444444█44444444444444444444444444444444444444444444444444444444444444444444444
555555555█5555555555555555555555555555555555555555555555555555555555555555555555
666666666666666█6666666666666666666666666666666666666666666666666666666666666666
7777777777777777█777777777777777777777777777777777777777777777777777777777777777
88888888888888888█88888888888888888888888888888888888888888888888888888888888888
999999999999999999█999999999999999999999999999999999999999999999999999999IBM5081
```

Table 198. Punched-card image of two numbers, +12345 and −67890

The technique of using the top two unnumbered "rows" of the card indicated both the end of a field of digits, and the sign of the number: if the number was positive, the 12-row (top row) of the card column containing the numeric digit punch was also punched, and if the number was negative the 11-row (the next to top row) was punched. This meant that +5 appeared as the letter "E" (with hex representation `X'C5'`), and −9 appeared as the letter "R" (with hex representation `X'D9'`). When "numbers" in this representation are read into memory as EBCDIC characters, the presence of a sign punch over the last digit automatically gives the zoned decimal representation of the number. (If a sign is prefixed to the digits, a + sign is a single punch in the 12-row, and a − sign is a single punch in the 11-row.)

Exercises

27.1.1.(2)+ In Figure 263 on page 461, why are **Numerics** and **Zones** defined with explicit length 8? Show the results if they had been defined instead by

```
Numerics DC    8X'0'
Zones    DC    8X'0'
```

27.1.2.(1)+ What will be the result of executing these two instructions?

```
         MVN   Target,Source
         MVZ   Target,Source
```

27.1.3.(3) Write an instruction sequence using TRT to test the validity of the digit and sign in the rightmost byte of the zoned decimal number at **ZTest**. If the digit portion is invalid, branch to **BadDigit**; if the sign portion is invalid, branch to **BadSign**; and if both are invalid, branch to **BadByte**. If you can also determine the sign of a valid byte, branch to **ZPlus** for a plus sign and to **ZMinus** for a minus sign.

27.1.4.(2)+ Write an instruction sequence using TRT to test the validity of all but the rightmost byte of the zoned decimal number at **ZTest**. If any byte is invalid, branch to **BadZDig** with GR1 containing the address of the invalid byte.

27.1.5.(1) In Table 197 on page 462, what will be printed for the rightmost byte of each item?

27.1.6.(2) Find the EBCDIC punched card code for the + and − characters. Now, do the same for the BCD punched card codes.

27.1.7.(1)+ After example 1 of using the MVZ instruction (on page 461), it was stated that the literals =5C'0' and =5C'9' could have been used. Explain why this is so.

27.1.8.(2) Consider the MVZ instruction in the same example as in Exercise 27.1.7. Could the statement have been written in any of the following forms? Explain why or why not.

```
MVZ   NDec,=XL5'FF'
MVZ   NDec,=(NDec)X'FF'
MVZ   NDec,=XL(L'NDec)X'FF'
MVZ   NDec,=(L'NDec)X'FF'
```

27.2. Zoned Decimal Constants

Zoned decimal constants are defined with a DC statement with type "Z". Each digit in the constant is translated into a single byte in storage and the Assembler assigns the preferred sign code to the rightmost byte. For example, the constants in Table 197 on page 462 could be defined with either of these statements:

```
ZoneA    DC    Z'12345',Z'-12345',Z'+003',Z'-0999'
ZoneB    DC    Z'12345,-12345,+003,-0999'
```

Figure 266. Zoned decimal constants with implied lengths

When the length of the constant is *implied* (as in Figure 266), the length of the constant is the same as the number of digits in the nominal value. As usual, the length attribute of the name-field symbol is that of the first constant in the first operand.

A decimal point may be placed anywhere in a zoned decimal constant. Its presence is ignored, and it does not appear in the generated constant.[175] Thus, both

```
         DC    Z'12345.'          and
         DC    Z'.12345'
```

generate X'F1F2F3F4C5', but their Integer and Scale attributes are 5 and 0, or 0 and 5, respectively. (See Exercise 27.2.8.)

To specify an explicit length, use a length modifier. If the value of the length modifier is less than the number of digits, the constant will be truncated at the left end to the required length:

```
         DC    ZL3'12345'    Truncated constant = X'F3F4C5'
```

If the value of the length modifier is greater than the number of digits, the Assembler will pad with high-order (zoned, EBCDIC) zeros, with representation X'F0':

```
         DC    ZL5'123'      Padded constant = X'F0F0F1F2C3'
```

The maximum length of a zoned decimal constant is 16 bytes.

The constants in Table 197 on page 462 could also be defined with explicit lengths:

```
ZoneC    DC    ZL5'12345',ZL5'-12345',ZL3'3',ZL4'-999'
```

Figure 267. Zoned decimal constants with explicit lengths

Scale and exponent *modifiers*, and decimal *exponents*, are not supported for Z-type constants.

175 The position of the decimal point is reflected in the integer and scale attributes of the symbol, which are useful mainly in macro instructions. They are rarely if ever used with zoned constants.

Exercises

27.2.1.(1)+ What data would be generated if we wrote the constants in Figure 266 on page 464 as

```
ZoneD    DC    ZL5'12345,-12345,+3,-999'  ?
```

Is anything different? If so, what and why?

27.2.2.(1) Show the generated data for these zoned decimal constants:

```
(1) Z'-1'    (2) Z'+0'   (3) Z'-042'   (4) ZL2'5'
```

27.2.3.(1) What decimal value is represented by the following zoned decimal constants?

```
(1) X'F0B9'  (2) X'F2'   (3) X'F0F0A7' (4) X'B0'
```

27.2.4.(1) Show the generated data for these zoned decimal constants:

```
(1) ZL16'123456.789'     (2) Z'123456789012345678'
(3) ZL5'.654321'         (4) ZL20'2.20'
```

27.2.5.(2)+ How many valid zoned decimal values can be represented in a single byte?

27.2.6.(2)+ What data would be generated if we wrote the constants in Figure 266 on page 464 as

```
ZoneD    DC    CL5'12345,-12345,+3,-999'  ?
```

What is different? Why?

27.2.7.(1)+ What are the length attributes of the symbols in Figure 266 on page 464?

27.2.8.(2)+ If a zoned decimal number is N digits long, write expressions relating N and the Length, Integer, and Scale Attributes.

27.3. Packed Decimal Representation

The packed decimal representation is often used when computing speed is less important than economy of storage space, and we want speed and simplicity of conversion to and from character form. Also, because humans calculate in decimal, operations like rounding are more intuitive than if the same calculation is done using binary data, and can be more accurate than the same computations in binary.

As its name implies, data is more closely “packed” than in the zoned decimal representation. The basic data element is the binary coded decimal (BCD) digit[176] represented by four bits; these BCD digits can be packed two to a byte. The six bit combinations corresponding to the hex digits A through F are *invalid data digits* in the packed decimal representation. If an invalid data digit appears in an arithmetic operation, the CPU will generate an interruption for invalid Decimal Data.

The rightmost digit position of a packed decimal number is reserved for its sign, which obeys the conventions shown in Figure 264 on page 462. As with zoned decimal data, the preferred values for the sign digit are X'C' for “+”, and X'D' for “−”.

Using the “pictorial” representation of Figure 265 on page 462, a packed decimal number will appear in storage as shown in Figure 268 on page 466. Note that there are no zone digits in a packed decimal number.

---

176 This representation, where a decimal *digit* is encoded in four bits, is often called “binary coded decimal”. To avoid confusion with the 6-bit *character* code that was also called “Binary Coded Decimal” (see Section 26.1.1 on page 429 and Table 170 on page 430), we will use the terms “packed decimal digit” or “decimal digit” instead.

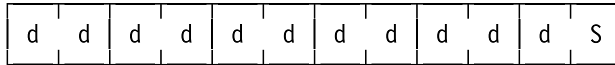


Figure 268. Representation of a packed decimal number

Packed decimal numbers look more like numbers we are familiar with, and the sign being at the right end is a small adjustment for us to make. Some examples are shown in Table 199.

Value	Representation
+12345	`X'12345C'`
-0012345	`X'0012345D'`
+3	`X'3C'`
-09990	`X'09990D'`
39	`X'039C'`

Table 199. Examples of packed decimal data

Any extra digits at the left end will be set to zero. Because the sign always occupies one digit position, an N-byte packed decimal number in memory always has an *odd* number of digits, 2N-1. Packed decimal data may be up to 16 bytes long, so it is possible to have 31-digit numbers; however, some operations on decimal data require shorter operands.

The packed decimal representation has an unusual feature: unlike two's complement binary, it is possible to have a negative sign digit and all zero data digits. Thus, packed decimal is a *sign-magnitude representation*. In most cases, a zero result of an operation is given a positive sign digit, but it is also possible to generate negative zeros in special cases. Thus it is *not* possible (as is possible in the two's complement binary representation) to determine that a number is strictly negative by examining its sign digit.

There is also an *unsigned* packed decimal representation used in conjunction with decimal floating point; we'll investigate it in Section 35.

Exercises

27.3.1.(2)+ A packed decimal number contains N digits; give a formula for the number of bytes required to hold it in memory.

27.3.2.(2) How many valid packed decimal values can be represented in a single byte?

27.3.3.(4) Test the packed decimal number at **Pack** for validity with a single TRT instruction and a suitable translate table. If the number is valid and nonzero branch to **PPlus** or **PMinus** depending on its sign. Otherwise,

- if the sign is invalid branch to **BadSign**
- if a byte contains an invalid digit branch to **BadDigit**
- if a byte contains both an invalid sign and digit branch to **BadByte**

with the address of the byte containing the invalid data in GR1.

27.3.4.(2)+ Suppose you must force the sign of the packed decimal number at **SomeVal** to be positive. Show you you can do this with one instruction.

27.3.5.(3) A packed decimal operand is known to have a bad numeric digit. Create a translate table that can be used with a TRT instruction to identify both the byte with the bad digit, and whether the first or second digit is invalid.

27.3.6.(3) Suppose you must force the sign of the packed decimal number at **SomeVal** to be negative. Can you do this with one instruction? (Compare to your answer for Exercise 27.3.4.)

27.4. Packed Decimal Constants

Packed decimal constants are defined using a DC statement with type “P”. For example, we could define the constants in Table 199 on page 466 with these statements:

```
PackA    DC    P'12345',P'-0012345',P'+3',P'-9990',P'39'
PackB    DC    P'12345',-012345,3,-09990,39'
```

Figure 269. Packed decimal constants with implied lengths

When the lengths are implied, the Assembler generates exactly the number of bytes needed to contain the constant, and no more. Remember that the number of packed decimal digits is always odd, so that a value with an even number of digits will have an extra high-order zero digit supplied by the Assembler. As with Z-type constants, the maximum length of a packed decimal constant is 16 bytes, or 31 decimal digits.

Explicit lengths are assigned to packed decimal constants in the usual way, with padding and truncation being performed at the left end of the constant. The constants in Figure 269 could be specified with explicit lengths, as shown in Figure 270.

```
PackC    DC    PL3'12345',PL4'-12345',PL1'3',PL3'-9990',PL2'39'
```

Figure 270. Packed decimal constants with explicit lengths

As with zoned decimal constants, scale and exponent modifiers are not allowed. An optional decimal point may be placed within a constant, but its presence is ignored in forming the constant. This means that we can write constants such as

```
YourPay  DC    P'947.24'          Stored value = 94724C
MyPay    DC    P'13.07'           Stored value = 01307C
```

and the decimal point can help you to understand an intended use of the data.

Because the position of the decimal point doesn't affect the generated constant, we can write constants like these, all of which generate the same data:

```
         DC    P'1307.'           Stored value = 01307C
         DC    P'130.7'           Stored value = 01307C
MyPay    DC    P'13.07'           Stored value = 01307C
         DC    P'1.307'           Stored value = 01307C
         DC    P'.1307'           Stored value = 01307C
```

Unfortunately, this means that *you* must remember where the decimal point lies. Programming financial applications involving fractional quantities like currency and interest rates in packed decimal can be quite difficult if you must know the position of the decimal point for the operands of each arithmetic operation.

Similarly, the same value can have multiple representations:

```
         DC    P'1307.'           Stored value = 01307C
         DC    PL4'1307'          Stored value = 0001307C
         DC    PL5'1307'          Stored value = 000001307C
         DC    PL6'1307'          Stored value = 00000001307C
```

We will see in Section 35 on page 680 that decimal floating-point arithmetic is much simpler.

27.4.1. Scale Attributes and Packed Decimal Constants (*)

Although a generated packed decimal constant ignores any decimal point in the nominal value, the Assembler assigns *Scale* and *Integer* Attributes to any symbol naming the constant. The value of the scale attribute is the number of decimal digits to the right of the decimal point. For example:

```
MyPay    DC    P'13.07'           Scale Attribute = 2, Integer Attribute = 2
MyPayA   DC    P'130.7'           Scale Attribute = 1, Integer Attribute = 3
MyPayB   DC    P'.0013'           Scale Attribute = 4, Integer Attribute = 0
MyPayC   DC    P'1.200'           Scale Attribute = 3, Integer Attribute = 1
MyPayD   DC    P'1307'            Scale Attribute = 0, Integer Attribute = 4
```

You can retrieve the scale attribute of a symbol using the S' operator, just as the L' operator retrieves its length attribute. Note that the sum of the Integer and Scale attributes is the number of digits in the constant. (See Exercise 27.4.15.)

Scale attributes can be very helpful in complex calculations involving numbers with fractional values, but the necessary techniques usually require some practice.

Exercises

27.4.1.(2)+ For each of the following zoned and packed decimal constants, state which ones are invalid, and why. For the valid constants, show (in hex) the generated bytes.

```
Z1       DC    Z'2147483648'
Z2       DC    ZL9'2147483647'
P3       DC    PL6'-99999999999'
P4       DC    P'+123,456,789'
Z5       DC    ZL20'500',PL20'500'
P6       DC    P'3.1415926535'
P7       DC    PL1'40',ZL1'40'
Z8       DC    Z'1,20,400,80000,1600000000'
```

27.4.2.(1)+ For each of the DC statements in Exercise 27.4.1, determine the length attribute of each symbol that names a valid constant.

27.4.3.(1)+ What sort of constant is generated by the DC operand PL2'−1000'? (Try it with the Assembler.)

27.4.4.(2) Both of these constants will be truncated, since more digits are specified than the explicit length allows.

```
         DC    PL2'0000437'
         DC    ZL3'-000904'
```

However, the truncated digits are all zeros; should this condition be considered an error by the Assembler?

27.4.5.(2) The nominal values of the constants **PackA** and **PackB** in Figure 269 on page 467 are different, but they generate the same machine language data. How is this possible?

27.4.6.(3) The packed decimal number at **P** is from 1 to 4 bytes long. Write instructions to "load" the number into GR6 to form a valid 4-byte packed decimal number. For example, if c(P)=X'123C', the result in GR6 should be X'0000123C'.

27.4.7.(3) Repeat Exercise 27.4.6, now assuming the packed operand at **P** can be from 1 to 8 bytes long, and that it should be "loaded" into GG6.

27.4.8.(2) A four-byte area of memory contains the bit pattern X'4040405C'. What is represented by that pattern?

27.4.9.(5) Write (and test!) a single DC statement that generates all 1000 two-byte packed decimal constants from X'000C' through X'999C', representing 000+ through 999+.

27.4.10.(5) Write (and test!) a single DC statement that generates all 1000 three-byte zoned decimal constants from X'F0F0C0' through X'F9F9C9'.

27.4.11.(1) Determine the scale attribute of each of these constants:

```
A        DC    P'1.4142135'
B        DC    P'002471.360'
C        DC    P'16777216.5'
D        DC    P'186.3541'
E        DC    P'2236067977'
```

27.4.12.(3) The Assembler assigns an *Integer Attribute* to symbols naming packed decimal data. Its value is the number of decimal digits to the left of the decimal point, and the value can be retrieved with the `I'` operator.

Given the `L'` (Length) and `S'` (Scale) attribute values of a symbol `X` naming a packed decimal constant, derive a formula for the value of its Integer Attribute.

27.4.13.(2) Using your results from Exercise 27.4.12, determine the Length, Integer, and Scale Attributes of the symbols in Exercise 27.4.11.

27.4.14.(2) Using the definitions in Exercise 27.4.12, given the `L'` and `S'` attribute values of a symbol `X` naming a *zoned* decimal constant, derive a formula for the value of its Integer Attribute.

27.4.15.(1)+ If a packed decimal number is N digits long, create expressions relating its Length, Integer, and Scale attributes.

27.5. Converting Between Packed and Zoned

Because packed decimal data is often used in applications needing conversion between external (character) and internal forms, System z provides two instructions that simplify this process.

The PACK instruction converts data from zoned to packed decimal, and the UNPK instruction converts packed decimal data to zoned. (There are two other powerful instructions for converting from packed decimal to character form, ED and EDMK; we'll discuss them in Section 30.) Both PACK and UNPK are SS-type instructions, as illustrated in Table 9 on page 53.

The Assembler Language syntax for these two instructions is shown in Figure 271:

$$\texttt{mnemonic}\quad D_1(N_1,B_1),D_2(N_2,B_2)$$

Figure 271. Format of typical two-length SS-type instructions

Compared to Figure 196 on page 366, which has only one length field, these have two, N_1 and N_2; and, unlike the SS-type instructions described in Section 25 (compare Table 152 on page 404), the length specification byte requires *two* four-bit length fields, as in Table 200.

opcode	L_1	L_2	B_1	D_1	B_2	D_2

Table 200. Format of two-length SS-type instructions

The Encoded Length digits L_1 and L_2 take values between 0 and 15, specifying operand lengths N_1 and N_2 of 1 to 16 bytes. (Section 24.1 on page 365 explains the reasons for the differences between N, N_1, N_2, and L, L_1, L_2: the “L” values the CPU sees are one less than the “N” values you specify.)

Because there are six operand-dependent components in these instructions, the operand field of the machine instruction statements may take a great variety of forms. Each operand may have the same form as the first operand of the one-length SS-type instructions discussed in Section 25; that is, *both* length *and* address may be implied or explicit for *each* operand.

The possible forms of either operand are summarized in Table 201 on page 470. Though the quantities S_1, B_1, D_1, and N_1 refer to the first operands, the possible forms of the second operand are identical.

	Implied Length	Explicit Length
Implied Address	S_1	$S_1(N_1)$
Explicit Address	$D_1(,B_1)$	$D_1(N_1,B_1)$

Table 201. Operands of two-length SS-type instructions

These examples illustrate the four forms of a first operand:

```
         PACK  A,B                 Implied address and length
         PACK  A(7),B              Implied address, explicit length
         PACK  24(,9),B            Explicit address, implied length
         PACK  24(7,9),B           Explicit address and length
```

The form of the first operand in the third example is rarely used, because the implied length of the absolute first operand will usually be 1, the length attribute of the term used for the displacement D_1.

If the length of an operand is implied, the value assigned to the Program Length Expressions N_1 and N_2 is the length attribute of the leftmost term in the expressions S_1, S_2, D_1 or D_2, as appropriate.

As we saw in Section 24 for the one-length SS-type instructions, the number the Assembler places into the Encoded Length digits L_1 and L_2 is actually *one less* than the value of the corresponding Program Length Expressions N_1 and N_2, whether implied or explicit. As with other SS-type instructions, an explicit Length Expression N of value zero is assembled as a zero length digit.

To illustrate, suppose the symbols **AA** and **BB** have length attributes 2 and 11, respectively. Then the following statements would be assembled as shown. Only the length digits L_1 and L_2 are important for this example; ignore the bddd base and displacement fields. (The operation codes X'F2' and X'F3' are for PACK and UNPK respectively, as shown in Table 196 on page 460.)

```
*        Instruction               Assembled Form
*
         PACK  AA(5),BB(5)         F244 bddd bddd  N1=5,N2=5   L1=4,L2=4
         UNPK  BB,AA               F3A1 bddd bddd  N1=11,N2=2  L1=10,L2=1
         PACK  0(16,9),65(,2)      F2F0 bddd bddd  N1=16,N2=0  L1=15,L2=0
         PACK  AA(0),BB(0)         F200 bddd bddd  N1=0,N2=0   L1=0,L2=0
         UNPK  BB-AA(,9),BB(L'AA)  F3A1 bddd bddd  N1=11,N2=2  L1=10,L2=1
         PACK  AA,BB+11(3)         F212 bddd bddd  N1=2,N2=3   L1=1,L2=2
         - - -
AA       DS    PL2
BB       DS    ZL11
```

Figure 272. Examples of assembled PACK and UNPK instructions

In each case the L value is one less than the N value, but if the N value is zero, the L value is also zero.

This illustrates another reason why the Encoded Length byte L (in one-length SS-type instructions), and the Encoded Length digits L_1 and L_2 (in two-length SS-type instructions), are one less than the values given for the Length Expressions in machine instruction statements. To obtain the addresses of the starting bytes of the operands of a PACK or UNPK instruction, the CPU simply adds the length specification digits to the Effective Addresses derived from the addressing halfwords.

The PACK and UNPK instructions, unlike many of the SS-type instructions we have discussed, process data from *right* to *left*. Neither sets the Condition Code.

Even though we're describing PACK and UNPK in the context of zoned and packed decimal data, the instructions are not sensitive to data types: they simply move data in prescribed ways, whatever their type.

Exercises

27.5.1.(1)+ In Figure 272 on page 470, determine the bddd values for every *explicit* address.

27.6. The PACK Instruction

The PACK instruction (and the UNPK instruction to be described in Section 27.7) have the machine instruction format shown in Table 200 on page 469, and the operands of its assembler instruction statements take any of the forms shown in Table 201 on page 470.

PACK converts data from zoned to packed form. Its operation is easily visualized by writing a number in both representations. We'll use +12345 again, which has these zoned and packed forms:

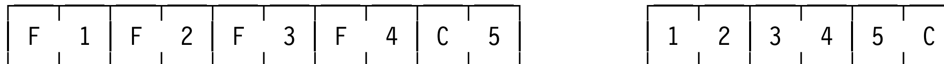


Figure 273. Zoned and packed forms of +12345

If we interchange the digits of the rightmost byte of the zoned form, we obtain the rightmost byte of the packed form. The remaining digits are extracted in right-to-left order from the numeric portion of each zoned byte, and placed into the proper positions in the packed operand. This is illustrated in Figure 274.

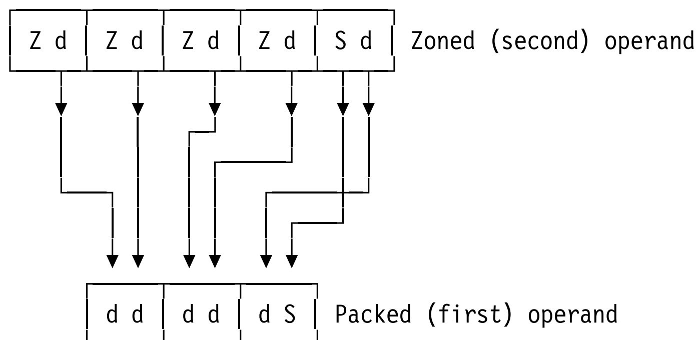


Figure 274. PACK instruction operation

If the zoned and packed operands were named **ZonOp** and **PackOp** respectively, we could perform the operation pictured in Figure 274 with this instruction:

```
         PACK  PackOp,ZonOp        From zoned to packed
         - - -
ZonOp    DC    Z'12345'            Zoned operand, length = 5 bytes
PackOp   DS    PL3                 Packed operand, length = 3 bytes
```

Figure 275. Converting from zoned to packed decimal using PACK

No checking is done for invalid data digits in the packed operand; the instruction just moves data as illustrated. But, you should ensure that valid packed operands are generated if they are to be used for arithmetic.

In the operation illustrated in Figure 274, the operands were chosen to contain exactly the right number of digits. However, either operand of a PACK instruction might be too long for the other. The rules for such cases are

1. If the first (packed) operand is completed before the zoned operand is exhausted, the rest of the zoned operand is ignored.

2. If the second (zoned) operand is exhausted before the packed operand is completed, the remaining high-order digits of the packed operand are filled with zeros. (If this was not done, the high-order digit positions of the packed operand might contain unknown values.)

The PACK operation is therefore controlled by the length of the first operand; the result *always* has the specified length. The second (zoned) operand is unmodified (assuming no overlap).

To illustrate, suppose we execute some PACK instructions with operands of various lengths, as in Figure 276. The result of each operation is shown in the comment field of the statement defining the first operand.

```
         PACK  P1,Zone(4)        Lengths 1, 4
         PACK  P2,Zone           Lengths 2, 5
         PACK  P3,Zone(2)        Lengths 3, 2
         PACK  P5,Zone           Lengths 5, 5
         - - -
Zone     DC    Z'12345'          Zoned operand = X'F1F2F3F4C5'
P1       DS    PL1               Result = X'4F'
P2       DS    PL2               Result = X'345C'
P3       DS    PL3               Result = X'00012F'
P5       DS    PL5               Result = X'000012345C'
```

Figure 276. Examples of the PACK instruction

Note especially the result at **P1**: because the length of the second operand in the first PACK instruction was explicitly specified as 4 (rather than the correct implied length of 5, as in the second PACK instruction), the rightmost byte of the second operand is X'F4' You must be careful to specify correct lengths for the operands of PACK instructions because significant data digits can easily be lost, or undesired or unexpected values may be generated.

The first and second operands of a PACK instruction could overlap. While such uses are almost always accidental, this programming trick might be useful in special cases.

- If overlap occurs with PACK operands, the CPU will store the completed first operand (packed) byte after fetching the next two second operand (zoned) bytes (or the next one byte, if that will exhaust the second operand).

We saw in Figure 274 on page 471 that the two hex digits of the rightmost byte are interchanged. We can use this property of PACK to swap the digits of any byte:

```
         PACK  A,A               Swap digits of A
         - - -
A        DC    X'12'             Result = X'21'
```

Figure 277. Digit swap using PACK

Exercises

27.6.1.(1)+ Suppose a zoned decimal operand containing NZ bytes is to be converted to packed decimal form. Give a formula for NP, the minimum number of bytes required for the packed decimal operand.

27.6.2.(2) Suppose you execute this PACK instruction:

```
         PACK  PData,XXX
         - - -
PData    DS    PL6
```

Show the contents of the field at **PData** if the data at **XXX** is defined as follows:

1. `XXX    DC    F'-9'`
2. `XXX    DC    C'Hello, World!'`
3. `XXX    DC    CA'Hello, World!'`

27.6.3.(4) Write a program segment which will simulate the action of the PACK instruction. Assume that the packed and zoned operands are stored at **POP** and **ZOP** respectively, and that the length attributes of those symbols are the correct operand lengths. That is, the code should produce the same result as:

```
         PACK  POP,ZOP           From zoned to packed
```

Without using a PACK instruction, of course!

27.6.4.(3) Suppose a PACK instruction specifies operands that overlap. Where must the right-most byte of the first (target) operand be placed relative to the first byte of the second (source) operand, so that the correct result will be obtained in all cases?

27.6.5.(2)+ For the zoned decimal constant named **ZData**, show the result of executing each of the following PACK instructions.

```
ZData    DC    Z'2468',Z'1357'
         - - -
         PACK  P1(2),ZData
         PACK  P2(3),ZData
         PACK  P3(1),ZData+1(2)
         PACK  P4(6),ZData+2(6)
         PACK  P5(5),ZData(L'ZData+3)
```

27.6.6.(3)+ Suppose a zoned decimal operand is packed "onto itself". That is, the first and second operands of a PACK instruction are identical. What is the maximum length of the operand such that the result will be correct? The minimum length?

27.6.7.(3) Give the contents of the storage area named **DATA** after executing the following PACK instruction.

```
         PACK  DATA,DATA
         - - -
DATA     DC    X'123456789ABCDEF'
```

27.6.8.(2)+ A common programming convention is that an all-blank field on a data record should be interpreted as a zero value. What will be the result of PACKing a string of blank characters? Need anything be done to the result? If so, what?

27.6.9.(2) Suppose we start with a zoned decimal operand at **Zone** having length N bytes, and execute the instruction

```
         PACK  PackOp,Zone(K)
```

where the length K is always less than or equal to N. Consider possible lengths for the first operand **PackOp**: under what conditions will a valid packed result be obtained?

27.6.10.(2) Suppose you need to pack some zoned data at **ZData** by EXecuting this PACK instruction:

```
         PACK  PackData(*-*),ZData
```

The length of the packed decimal field at **PackData** is contained in GR2, and does not exceed 16. Write a sequence of instructions to do this.

27.6.11.(1)+ Suppose you execute this PACK instruction:

```
         PACK  P,P               Pack a field onto itself
         - - -
P        DC    X'ABCDEF'
```

What result will be at **P**?

27.6.12.(2) Suppose you execute this PACK instruction:

```
         PACK  Q,P               Pack a field onto itself
         - - -
Q        DS    XL3               Result field
         ORG   Q+1               Source field starts at Q+1
P        DC    X'ABCDEF'         Source field
```

What result will be at **Q**?

27.6.13.(2) Suppose you execute this PACK instruction:

```
         PACK  Q,P               Pack a field onto itself
         - - -
Q        DS    XL3               Result field
         ORG   Q-1               Source field starts at Q-1
P        DC    X'ABCDEF'         Source field
```

What result will be at **Q**?

27.6.14.(1)+ When you PACK a zoned decimal field, what will be different in the result if the numeric digits are preceded by blanks or by leading zeros?

27.6.15.(2)+ Suppose you execute this PACK instruction

```
         PACK  Packed,Source
```

for each of the following **Source** constants. In each case, show the result at **Packed**, and state whether the result is or is not a valid packed decimal number, and if not, why it is invalid.

```
(1)   Source   DC    C'34567'
(2)   Source   DC    C'ABCDE'
(3)   Source   DC    C'****0'
(4)   Source   DC    C'$2.98'
(5)   Source   DC    C'     '
(6)   Source   DC    C'VWXYZ'
```

27.7. The UNPK Instruction

The UNPK instruction performs the inverse of PACK: it transforms a packed decimal operand into zoned decimal form. Like PACK, the UNPK instruction

- does not set the Condition Code,
- does not check for valid data or sign digits in the packed operand, and
- is controlled by the length of the first (zoned) operand.

The digits of the low-order byte of the second (packed) operand are switched, and the result becomes the rightmost byte of the first (zoned) operand. Successive digits are then taken in right-to-left order from the packed operand and placed into the numeric-digit positions of successive bytes in the zoned operand, with X'F' placed in each zone digit.

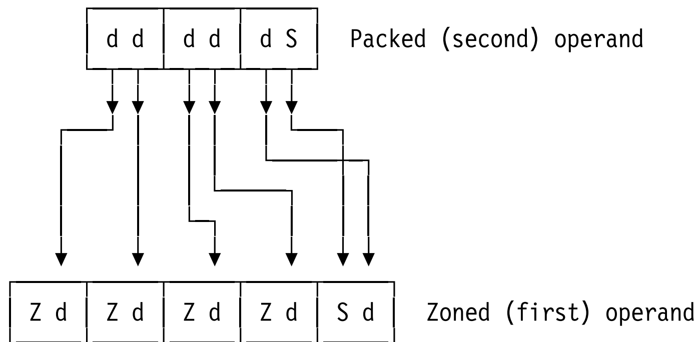


Figure 278. Operation of the UNPK instruction

Figure 279 shows how to convert the packed value +12345 to zoned form.

```
         UNPK  ZonOp,PackOp        From packed to zoned
         - - -
ZonOp    DS    ZL5                 Zoned operand, length = 5 bytes
PackOp   DC    P'12345'            Packed operand, length = 3 bytes
```

Figure 279. Example of an UNPK instruction

In the PACK operation, the zone digits of the zoned operand were discarded. For the UNPK operation, the zone digits are supplied by the CPU.[177]

When the lengths of the two operands do not correspond exactly, rules like those for PACK apply to UNPK:

1. If the first (zoned) operand is completed before the packed operand is exhausted, the rest of the packed operand is ignored.
2. If the second (packed) operand is exhausted before the zoned operand is completed, the remaining high-order bytes of the zoned operand are completed with zoned zeros.

To illustrate the operation of the UNPK instruction, the result of unpacking a packed decimal number is shown in the comment field of each statement.

```
         UNPK  Z5,Pack             Data lengths 5, 3
         UNPK  Z7,Pack             Data lengths 7, 3
         UNPK  ZB,Pack             Data lengths 3, 3
         UNPK  ZA,Pack(2)          Data lengths 3, 2
         UNPK  Z4,Pack(1)          Data lengths 4, 1
         - - -
Pack     DC    P'12345'            3-byte packed operand, X'12345C'
Z5       DS    ZL5                 Result = X'F1F2F3F4C5'
Z7       DS    ZL7                 Result = X'F0F0F1F2F3F4C5'
ZB       DS    ZL3                 Result = X'F3F4F5'
ZA       DS    ZL3                 Result = X'F1F243'
Z4       DS    ZL4                 Result = X'F0F0F021'
```

Figure 280. Examples of UNPK instructions

If you study the results at **ZA** and **Z4** you will understand the need for care in specifying the lengths of the operands, since possibly invalid zoned decimal results can be generated; we saw similar incorrect results in Figure 276 on page 472.

[177] On System/360 CPUs, the zone digits were determined by the setting of the "A" bit in the PSW, which is no longer used. If the "A" bit was zero, the CPU assumed that EBCDIC characters are desired, and automatically supplied a zone digit X'F' where needed; if the "A" bit was 1, the CPU assumed that characters in the USASCII representation are desired, and supplied X'5' for the zone digits. (This is different from the current definition of ASCII, where the zone digits are X'3'.)

As was done with the PACK instruction in Figure 277 on page 472, we can use UNPK to swap the hex digits of a byte:

```
         UNPK  A,A                Swap digits of A
         - - -
A        DC    X'12'              Result = X'21'
```

Figure 281. Digit swap using UNPK

The first and second operands of an UNPK instruction could overlap. While such uses are almost always accidental, this behavior might be useful in special cases.

- If overlap occurs with UNPK operands, the CPU processes the operands by storing two result bytes immediately after the necessary source operand byte has been fetched for the next step.

In Section 30.3 on page 536 we'll see how the powerful ED and EDMK instructions make it easy to format packed decimal data for printing and display.

Exercises

27.7.1.(1)+ In Figure 275 on page 471, what will be the contents of the memory area named **PackOp** after executing the PACK instruction? In Figure 279 on page 475, what will be at **ZonOp** after executing the UNPK instruction?

27.7.2.(2) Suppose the CPU executes this UNPK instruction:

```
         UNPK  ZData,YYY
         - - -
ZData    DS    ZL9
```

Show the contents of the field at **ZData** if the data at **YYY** is defined as follows:

```
 1. YYY     DC    F'-9'
 2. YYY     DC    C'Hello, World!'
 3. YYY     DC    CA'Hello, World!'
```

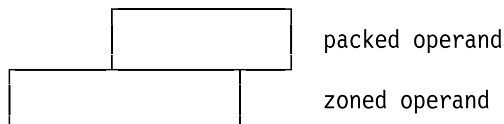
27.7.3.(3) Determine the result of executing an UNPK instruction instead of the PACK instruction in Exercise 27.6.7.

27.7.4.(4) Write a program segment like that in Exercise 27.6.3, except that the instruction

```
         UNPK  ZOP,POP            From packed to zoned
```

should be simulated.

27.7.5.(4) Suppose the operands of an UNPK instruction overlap, and assume that the address of the last byte of the packed decimal operand is greater than or equal to the address of the last byte of the zoned operand. What rule or rules can you state that will guarantee the same results as if there were no operand overlap? For example, in this sketch the address of the last byte of the packed operand is greater than the address of the last byte of the zoned operand.



27.7.6.(2)+ For the given packed decimal constant below, show the result of executing each of the following UNPK instructions.

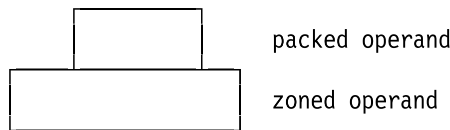
```
PData    DC    P'123456'
         - - -
         UNPK  Z1(6),PData
         UNPK  Z2(7),PData
         UNPK  Z3(2),PData
         UNPK  Z4(6),PData+2(2)
         UNPK  Z5(4),PData(3)
         UNPK  Z6(2),PData+3
```

27.7.7.(3) Do the same as in Exercise 27.6.6, but now consider unpacking a packed decimal operand "onto itself" using an UNPK instruction with identical operands.

27.7.8.(2) Suppose a packed decimal operand NP bytes long is to be converted to zoned decimal form. Give a formula for NZ, the minimum number of bytes required for the zoned operand. Is your formula the same (aside from algebraic rearrangement) as the result you obtained in Exercise 27.6.1? If not, why not?

27.7.9.(3) In the PACK and UNPK instructions, the operation is controlled by the length specification digit for the first operand, namely L_1. Why not use L_2? What would happen if the longer length were chosen? The shorter?

27.7.10.(5) In Exercise 27.7.5 you considered the situation where the operands of an UNPK instruction overlap, and the address of the rightmost byte of the second (packed) operand was not less than the address of the first (zoned) operand. Now, suppose the operands overlap, but the address of the rightmost byte of the packed operand is *less* than the address of the rightmost byte of the zoned operand: what rule or rules can you state for the relations between operand lengths and addresses that will guarantee the same results as if there were no operand overlap? For example, in this sketch the address of the last byte of the packed operand is less than the address of the last byte of the zoned operand.



27.7.11.(4) Repeat Exercise 27.7.3, but now assume you execute the UNPK instruction twice in succession. What will be in the field named **Data** when the second instruction completes execution?

27.7.12.(2) Figure 276 on page 472 and Figure 280 on page 475 illustrate how invalid results can be generated when the length supplied for the second operand of the PACK and UNPK instructions is not the same as the length of the datum. Give a general guideline which will generate a valid result in such cases.

27.7.13.(3) Write program segments to perform the functions described in Exercises 27.7.10 and 27.7.11 using shifts, loops, and no translate instructions.

27.7.14.(2) Suppose you execute this UNPK instruction:

```
         UNPK  U,P                Unpack P operand to U
         - - -
U        DC    XL6'0'             U initialized to 6 zero bytes
         Org   U+2                Position P at third byte of U
P        DC    X'123456'          Three bytes of second operand P
```

What result will appear at **U**?

27.7.15.(3) Suppose you execute this UNPK instruction:

```
         UNPK  U,P               Unpack P operand to U
         - - -
U        DC    XL6'0'            U initialized to 6 zero bytes
         Org   U+3               Position P at fourth byte of U
P        DC    X'123456'         Three bytes of second operand P
```

What result will appear at **U**?

27.7.16.(3) Suppose you execute this UNPK instruction:

```
         UNPK  U,P               Unpack P operand to U
         - - -
U        DC    XL6'0'            U initialized to 6 zero bytes
         Org   U+4               Position P at fifth byte of U
P        DC    X'123456'         Three bytes of second operand P
```

What result will appear at **U**?

27.7.17.(2)+ If you execute this UNPK instruction, what will be the result at **Answer**?

```
         UNPK  Answer,Start
         - - -
Answer   DS    ZL7
Start    DC    P'76543'
```

27.8. Packing and Unpacking ASCII and Unicode Data (*)

Unlike the PACK and UNPK instructions, the packed decimal operand of the last four instructions in Table 196 on page 460 is always 16 bytes long, and the Length Expression of the instruction gives the length of the "zoned" operand. However, ASCII and Unicode numeric characters are not "zoned" in the same sense as zoned decimal data; there is no special sign code in the rightmost byte.

Overlapping operands always generate unpredictable results.

As with PACK and UNPK, the ASCII pack and unpack instructions are not sensitive to data types; they simply move data fields in a prescribed way. Both process data from right to left.

27.8.1. Packing ASCII and Unicode Data

The PKA and PKU instructions convert numeric characters to packed decimal format. PKA is simpler: like PACK, it extracts the numeric digits from the second operand and packs them into the first operand. Both PKA and PKU have the format shown in Table 202; note that the Encoded Length L refers to the *second* operand. Because there is no "zone" sign on the low-order character, the CPU automatically inserts a `X'C'` plus sign. It's up to you to know whether the value should actually have a minus sign.

This format is the same as many of the SS-type instructions we've seen. However, the assembler instruction statement format is different: the Length Expression N is specified in the *second* operand:

```
mnemonic  D1(B1),D2(N,B2)
```

Neither PKA nor PKU changes the Condition Code.

opcode	L	B_1	D_1	B_2	D_2

Table 202. Format of PKA and PKU instructions

Figure 282 on page 479 shows an example of packing the ASCII characters at **AChars** into 16 bytes starting at **PDecA**.

```
        PKA   PDecA,AChars
        - - -
PDecA   DS    PL16               Packed decimal result (16 bytes)
AChars  DC    CA'1234567890123'  ASCII numeric characters (13 bytes)
* Result at PDecA = X'000000000000000000001234567890123C'
```

Figure 282. Packing ASCII characters

If the zoned operand has too few digits to fill all 31 digit positions of the packed operand, the remaining high-order digits are set to zero, as in Figure 282 where the ASCII operand contains only 13 digits.

PKU operates in much the same way. Since "Unicode" means UTF-16 here, the "zoned" operand is pairs of bytes. Each numeric digit is extracted from the rightmost four bits and placed into the packed operand. For example, if the Unicode operand is the two UTF-16 characters 47, or X'00340037', the packed decimal result would contain X'00...0047C'. Figure 283 shows how you could pack Unicode characters:

```
        PKU   PDecU,UChars       Pack Unicode characters
        - - -
PDecU   DS    PL16               Packed decimal result (16 bytes)
UChars  DC    CU'1234567890123'  Unicode numeric characters (26 bytes)
* Result at PDecU = X'000000000000000000001234567890123C'
```

Figure 283. Packing Unicode characters

To avoid the possibility that the zoned operand could be too long, its length is limited:

- For PKA, the zoned operand length N may be at most 32 bytes ($0 \le L \le 31$)
- For PKU, the zoned operand length N may be at most 64 bytes ($0 \le L \le 63$)

It may seem strange to allow the zoned operand to contain one more character than will fit into the 31-digit packed operand, but if the maximum length is specified the CPU simply ignores the leftmost "zoned" character.

27.8.2. Unpacking ASCII and Unicode Data

The UNPKA and UNPKU instructions convert packed decimal data to ASCII and Unicode characters, respectively. Like PKA and PKU, the packed operand is always 16 bytes long; and unlike them, these unpack instructions set the Condition Code — but in a rather strange way, as we'll see.

The Assembler Language syntax of these two instructions is

        mnemonic $D_1(N,B_1),D_2(B_2)$

where the Length Expression N is part of the *first* operand.

The length field of UNPKA and UNPKU holds L, the Encoded Length of the first operand, as shown in Table 203:

opcode	L	B_1	D_1	B_2	D_2

Table 203. Format of UNPKA and UNPKU instructions

As with PKA and PKU, the length of the first operand is limited:

- For UNPKA, the zoned operand length N may be at most 32 bytes (so $0 \le N \le 32$ and $0 \le L \le 31$).
- For UNPKU, the zoned operand length N may be at most 64 bytes (so $0 \le N \le 64$ and $0 \le L \le 63$).

Because there is no “zoned sign” for ASCII and Unicode characters, the sign digit of the packed operand is used only to set the Condition Code. Instead of the EBCDIC X'F' zones inserted by UNPK, the CPU inserts X'3' digits for UNPKA, and inserts X'003' digits for UNPKU.

The unpacking operation proceeds from right to left, and is controlled by the length of the first operand, so portions of the packed second operand may be ignored. For example:

```
         UNPKA AChars,PDec        Convert to ASCII characters
         UNPKU UChars,PDec        Convert to Unicode characters
         - - -
PDec     DC    PL16'98765432'     31 packed decimal digits
AChars   DS    CL12               Space for 12 ASCII characters
UChars   DS    CL24               Space for 12 Unicode characters
```

Figure 284. Unpacking to ASCII and Unicode characters

In this case, the results at **AChars** and **UChars** will be

```
         X'30 30 30 30 39 38 37 36 35 34 33 32'
and
         X'0030 0030 0030 0030 0039 0038 0037 0036 0035 0034 0033 0032'
```

where spaces were inserted for readability.

However, if the first operand field is not long enough to contain all significant digits, no indication is given.

```
         UNPKA AChar2,PDec        Convert to ASCII characters
         UNPKU UChar2,PDec        Convert to Unicode characters
         - - -
PDec     DC    PL16'98765432'     31 packed decimal digits
AChar2   DS    CL7                Space for 7 ASCII characters
UChar2   DS    CL14               Space for 14 Unicode characters
```

In this case, the results at **AChar2** and **UChar2** will be

```
         X'38 37 36 35 34 33 32'
and
         X'0038 0037 0036 0035 0034 0033 0032'
```

The Condition Code settings after UNPKA and UNPKU are given in Table 204.

CC	Meaning
0	Packed operand sign is + (X'A, C, E, F')
1	Packed operand sign is − (X'B, D')
3	Packed operand sign is invalid.

Table 204. CC settings after UNPKA, UNPKU instructions

Warning

Every other instruction that tests a numeric result sets the Condition Code to zero for a zero result. These two instructions set Condition Code zero for a *positive* operand. If your program depends on the CC setting after these instructions, be *very* careful.

Exercises

27.8.1.(2) Show the generated packed decimal data at **APack** after you execute this PKA instruction:

```
         PKA   APAck,AChars
         - - -
AChars   DC    CA'1234567890ABCDEFGHIJKLM'
APack    DS    PL16                Packed operand
```

(See the ASCII encodings in Table 172 on page 433.)

27.8.2.(2) What will be the result at **PA** of this PKA instruction?

```
         PKA   PA,A(30)
         - - -
A        DC    10X'ABCDEF'         Zoned operand
PA       DS    PL16                Packed operand
```

27.8.3.(2) What will be the result at **PU** of this PKU instruction?

```
         PKU   PU,U(60)
         - - -
U        DC    20X'ABCDEF'         Zoned operand
PU       DS    PL16                Packed operand
```

27.8.4.(2) In Figure 282 on page 479, what will happen if **PDec** is shorter than 16 bytes?

27.8.5.(2)+ How should the Condition Code settings after UNPKA and UNPKU have been set?

27.8.6.(2) In Figure 284 on page 480, show the representation of the packed decimal operand at **PDec**.

27.8.7.(2)+ In Figure 284 on page 480, what will be the Condition Code setting after the UNPKA and UNPKU instructions?

27.9. Printing Hexadecimal Values

Beside its uses in converting from packed to zoned decimal, the UNPK instruction can also help convert a string of hexadecimal digits to “spread hex”, the EBCDIC characters which represent them. (See Figures 207 and 208 on page 381.) Suppose the 8 hex digits in the 4 bytes in memory at **Data** are to be converted to the 8-byte character string at **Chars**. Now, we cannot immediately write something like this:

```
         UNPK  Chars(8),Data(4)   Spread hex, first (incorrect) attempt
         - - -
Chars    DS    CL8                Space for character result
Data     DC    X'1234ABCD'        Initial data
```

Figure 285. Unpacking hex digits (incorrectly)

because **Chars** would contain

```
         X'F0F1F2F3F4FAFBDC'
```

The two digits in the rightmost byte of the second operand (at Data+3) were simply switched and placed in the byte at Chars+7; the remaining six digits were given zones, and the high-order character contains X'F0'. But X'FAFBDC' does not correctly represent the hex digits “ABCD” in character form.

To solve these problems, we supply an *extra* byte after each operand.

```
         UNPK  Chars(9),Data(5)    Spread hex, second (correct) attempt
         - - -
Chars    DS    CL8,C               Space for result and 1 extra byte
Data     DC    X'1234ABCD'         Data to be unpacked
         DS    X                   The extra “source” byte, X'xy'
```

Figure 286. Unpacking hex digits (correctly)

Now, **Chars** will contain

```
X'F1F2F3F4FAFBFCFD',X'yx'
```

where the rightmost (extra) byte contains uninteresting data of some sort, as indicated by X'yx'. These two digits are the switched digits from the byte at **Data**+4, following the right end of the bytes at **Data**.

The result shown is not yet in EBCDIC form, because the characters generated for the hex digits A through F have incorrect zones and numerics. For example, the hex digit A produces X'FA' instead of X'C1'. To complete the conversion, these bytes can be converted using a TR instruction (described in Section 24.9 on page 379). We don't need a 256-byte translate table because *all* the result bytes have X'F' zones. We need only a *16-byte* table, as shown in Figure 287.

```
         UNPK  Char(9),Data(5)         Convert to partial EBCDIC
         TR    Char(8),TRTab-X'F0'     Translate to true EBCDIC
         - - -
Char     DS    CL8,CL1                 Space for result and a junk byte
Data     DC    F'-5026'                Data to be spread
TRTab    DC    C'0123456789ABCDEF'     Translate table
```

Figure 287. Converting hex data to printable characters

This technique is used in many situations when diagnostic information is printed in hexadecimal.

Exercises

27.9.1.(2)+ In Figure 287, why is the second operand of the TR instruction written TRTab-X'F0'?

27.9.2.(2)+ In Figure 287, what will appear in the “extra byte” at Char+8 after the instructions are executed?

27.9.3.(3)+ Using the technique illustrated in Figure 287, convert the 16 hex digits in the doubleword at **DW** to a string of 16 EBCDIC characters representing the original value.

27.9.4.(2) Explain the appearance of the quantity X'F0' in the leftmost byte of the result shown following Figure 285 on page 481.

27.9.5.(3) In Figure 287, we assumed that the expression TRTab-X'F0' in the second (TR) instruction is addressable. What should be done if it is not?

27.9.6.(5) Suppose you want to print the contents of a byte as eight EBCDIC zero and one characters representing the bits. (This could be called “spread binary”.) For example, a byte containing X'5B' would be converted to the eight characters 01011011. The technique used in Section 27.9 for printing hexadecimal values can be extended, so that instead of a single UNPK and TR, you can use three UNPK instructions and two TR instructions. Use a translate table whose last 16 bytes contain

```
X'00010405101114154041444550515455'
```

First, work through an example to show that your method works for all byte values. Then, write a instruction sequence a to convert the byte at **Byte** to eight “spread binary” characters starting at **BinaryCh**.

27.9.7.(5) Suppose you want to convert strings of eight EBCDIC ones and zeros to a bit pattern in a single byte, as represented by the characters. For example, the characters 01011011 would be converted to X'5B'. Write an instruction sequence using three PACK and two TR instructions which will convert the eight characters at **BinaryCh** to a single byte at **Byte**. The translate table needed is 86 bytes long, and not all of it contains necessary data.

27.10. Summary

The move instructions we've discussed in this section are summarized in Table 205.

Function	Numeric digits	Zone digits
Move	MVN	MVZ

Table 205. Instructions for moving numeric and zone digits

The packing and unpacking instructions we've discussed in this section are summarized in Table 206.

Function	Zoned EBCDIC	Zoned ASCII	Zoned Unicode
Pack	PACK	PKA	PKU
Unpack	UNPK	UNPKA	UNPKU

Table 206. Instructions for packing and unpacking data

Terms and Definitions

ASCII numeric characters
ASCII characters with representations between X'30' and X'39'.

numeric digit
The rightmost 4 bits of a byte.

preferred sign code
For packed and zoned decimal numbers, there are six valid sign codes: X'A', X'C', X'E', and X'F' indicate +, and X'B' and X'D' indicate −. The preferred codes are X'C' and X'D'; these are the sign codes generated by packed decimal arithmetic operations.

Unicode numeric characters
Unicode characters with representations between X'0030' and X'0039'.

zone digit
The leftmost 4 bits of a byte.

Programming Problems

Problem 27.1.(2)+ Write a program that will "dump" itself, *without* using the DUMPOUT macro. That is, the program should print the hexadecimal contents of each byte in the area of memory it occupies.

Problem 27.2.(2) Write a program that will read hexadecimal "numbers" as EBCDIC characters in "spread hex" from 80-byte records, and convert them to internal hex values. Store them in a table, and then dump out the table area to verify your conversions. For example, if two characters of an input record are C1, the corresponding table entry will be the byte X'C1'.

28. Packed Decimal Arithmetic

```
 2222222222     8888888888
222222222222   888888888888
22        22   88        88
          22   88        88
          22    88      88
         22      88888888
       22        88888888
     22         88      88
   22          88        88
 22            88        88
222222222222   888888888888
222222222222    8888888888
```

In this section we will examine the methods used by the CPU to perform packed decimal arithmetic.

Because we are all familiar with decimal arithmetic, it may seem strange to investigate the way it is done by the CPU. While the broader features of decimal arithmetic are simple, its implementation on a binary machine leads to some unexpected subtleties.

In this and the following sections, we will use a simple notation for numbers in the packed decimal representation. Instead of "−123" we will write "123−", because this form is closer to the internal decimal representation of −123, namely X'123D'. Thus, if a number is written with a trailing sign, its representation in memory can be visualized by substituting the hex digits C and D for the + and − signs, respectively.

For convenience, we will refer to the packed decimal representation simply as "decimal" whenever "packed decimal" is clearly meant.

Writing zoned and packed decimal values

Because the sign of zoned and packed decimal numbers is always in the rightmost byte, it is convenient write such values with the sign on the right, as in 12345+ or 09990−.

28.1. General Rules

Before examining decimal arithmetic, we will state some general rules applying to decimal operations and operands.

1. The result of a decimal arithmetic operation (other than comparison) *always* replaces the first operand, so it has the same length as the first operand. (Packed decimal division fits both the quotient and the remainder in the first operand field.) Neither the second operand in memory (assuming no overlap) nor the contents of the General Registers are modified.

2. For System z processors, the preferred EBCDIC signs (X'C' and X'D') are always attached to results.[178]
3. Because decimal operands have finite length, results might overflow the allotted space: a decimal overflow exception will occur, and the Condition Code is set to 3.

   By setting a bit in the Program Mask (described in Section 16.2.1 on page 234), you can direct the CPU to ignore the exception condition for a decimal overflow. If the interruption *does* occur, the Interruption Code is set to 10 (X'000A'); the CC is still set to 3.
4. Because each decimal digit is represented by four bits, the CPU must guard against the possibility that a decimal digit position might contain one of the six invalid bit configurations. Similarly, a numeric digit may not occur as the lowest order hex digit (the sign digit) of an operand. If either error condition is detected, a *Data Exception* interruption occurs, and the Interruption Code is set to 7.[179]

   This interruption might seem to be an unnecessary nuisance, but it is actually *very* helpful. Almost all other arithmetic operations don't (or can't) check for invalid operands, and can generate meaningless results from invalid data. Because packed decimal operands are validated, your program is more likely to produce valid results, and can take corrective action when it's needed.

28.1.1. Precision and Accuracy

Packed decimal values are precise, because all values are integers. Their *accuracy* depends on the validity of the data used for each operation, and the care taken to preserve the inherent accuracy of the data. Inadequate precision can degrade accuracy, so be sure to specify enough digits so that significant digits won't be lost.

Exercises

28.1.1.(1) If you execute these instructions, what result is stored at **PWord**? Is it a valid packed decimal number? If so, what is its value?

```
         XR    1,1              Set GR1 to zero
         AHI   1,12             Add 12
         ST    1,PWord          Store the result
         - - -
PWord    DS    F
```

28.2. Decimal Addition and Subtraction

Because the CPU works from right to left in adding or subtracting decimal operands, excess digits are lost at the left, or high order, end of the operand.

The rules for adding and subtracting numbers in the packed decimal representation are the usual and familiar rules of arithmetic. There are three minor matters to be considered:

1. If the second operand is shorter than the first, then during its internal arithmetic the CPU will *extend* the second operand with enough high-order zeros so that its length matches that of the first operand. These extra zero digits are generated internally during the operation; the second operand in memory is never modified.
2. A decimal overflow condition occurs after an addition or subtraction if the first operand field is too short to hold all significant digits of the result. This can be due to one of two causes:

[178] In System/360, the USASCII sign digits (X'A' and X'B') were used if the “A” bit in the PSW was 1.

[179] The Data Exception Code (DXC) is also set to X'00'. For now, the DXC is unimportant; we'll see more about it in Section 35 when we discuss decimal floating-point.

- If one operand is longer than the other, high-order zeros are supplied internally to extend the shorter operand. For example, 3+ added to 999+ becomes 003+ + 999+, and the sum 1002+ overflows either operand used to hold the sum.
- If the resulting sum or difference causes a nonzero digit or a carry to be lost because the first operand field is too short, an overflow has occurred. For example, 6+ added to 7+ yields 13+ which overflows either operand used to hold the sum.

3. The CPU always attaches a + sign to a valid *non-overflowed* zero result of addition or subtraction.
   - The resulting sign might be − in case of overflow! If the rightmost portion of an *overflowed* result has a negative sign and enough low-order zero digits to fill the first operand field, the CPU will generate a negative zero result. For example, adding two one-byte operands such as (5−)+(5−) will yield 0−, and a decimal overflow exception is indicated.

The Condition Code is set to reflect the status of the result, as shown in Table 207.

CC	Indication
0	Result is zero
1	Result is less than 0
2	Result is greater than 0
3	Decimal overflow

Table 207. CC settings for decimal addition and subtraction

To illustrate these rules, suppose we add the following two operands:

```
  00006+    (1st operand)
+ 23497+    (2nd operand)
```

The result, 23503+, replaces the first operand. If the two operands were

```
  23497+    (1st operand)
+     6+    (2nd operand)
```

then the result would be the same. But if the two operands had been defined as

```
    006+    (1st operand)
+ 23497+    (2nd operand)
```

then the result at the first operand location would be 503+, and an overflow exception would indicate the loss of significant digits due to truncation. Unlike adding binary integers in the general registers, decimal addition or subtraction overflow can depend on the order of the operands.

Be Careful!

The results of packed decimal addition may depend on the order of the operands.

Addition of two operands of unlike sign follows the same rules. For example, suppose we add 04006+ and 01005−, the result would be 03001+ as expected. (We will see shortly that this result, while expected, is not as easy to obtain). Suppose now that we wish to add these quantities:

```
    006+   (1st operand)
+ 01005-   (2nd operand)
```

The result, 999−, replaces the first operand. Even though there are nonzero digits in the second operand corresponding to the “internal extension” digit positions in the first operand, no overflow occurs because the result has become short enough to fit into the first operand field.

The rule for subtraction of decimal operands is simple: invert the sign of the second operand, and then add. The change of sign is done internally by the CPU during the subtraction; the second operand is never modified in memory.[180] Thus, the subtraction operation

```
  04006+   (1st operand)
- 01005-   (2nd operand, to be subtracted)
  05011+
```

is internally identical to the *addition* operation

```
  04006+   (1st operand)
+ 01005+   (2nd operand, negated and added)
  05011+
```

As noted in Section 16.2.1, we can use the SPM instruction to mask off a decimal overflow interruption. The result will be the same whether or not an interruption occurs, and the CC will be set to 3.

Exercises

28.2.1.(1)+ In Section 2.10, we saw that no overflow was possible in adding fixed-point binary operands if their signs differed. Why is it possible to cause a decimal overflow exception when adding decimal operands of unlike sign?

28.2.2.(2)+ For each of the following pairs of numbers, the first operand is given first. Determine (1) the resulting sum or difference, (2) the resulting CC setting, and (3) whether or not an overflow condition will occur.

(1) (2147483647+) + (2147483648+)
(2) (99999+) + (00002+)
(3) (99999−) + (00002+)
(4) (7−) − (97+)
(5) (45+) − (66−)
(6) (4−) − (4−)
(7) (745−) + (255−)
(8) (2+) − (99999+)

28.2.3.(3)+ Each of the following pairs of numbers is taken from the same operand in memory; that is, the low-order bytes of both operands coincide. Determine in each case (1) the resulting sum or difference, (2) the CC setting, and (3) whether or not an overflow condition will occur.

(1) (12345+) + (345+)
(2) (345+) − (12345+)
(3) (000−) + (0−)
(4) (729+) + (729+)
(5) (476692543−) − (6692543−)
(6) (12345+) − (12345+)

28.3. Decimal Comparison

Comparing two decimal operands is simply an internal subtraction; the setting of the Condition Code indicates the result, as shown in Table 208 on page 488. As in addition and subtraction, all digits and signs are checked for validity.

[180] If the operands overlap, either could be modified.

CC	Meaning
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

Table 208. CC setting after decimal comparison

The sign of a zero operand doesn't matter: 0− is considered equal to 0+.

Comparison is actually somewhat simpler than subtraction, because all arithmetic is internal. As many high-order zeros are supplied for the operands as necessary, and no overflow condition is recognized. Thus, comparing 99999+ and 2− leads to an *internal* subtraction

```
  99999+    (1st operand)
-     2-    (2nd operand)
```

which becomes, after adding extra digit positions and making the necessary sign changes, an addition operation:

```
  0099999+    (1st operand)
+ 0000002+    (2nd operand)
```

Because the internal result is positive, the CC will be set to 2 to indicate that operand 1 is greater than operand 2. This result would *not* have been obtained by directly subtracting the two operands (see Exercise 28.3.1).

Exercises

28.3.1.(1) Give two reasons why the result of comparing 99999+ and 2- would not have been obtained by performing a subtraction instead (aside from the fact that one operand would be modified).

28.3.2.(1) Suppose you compare these two packed decimal operands using a CLC instruction:

```
         CLC   PDP,PDM           Compare operands
         - - -
PDP      DC    P'+0'
PDM      DC    P'-0'
```

What will be the resulting CC setting?

28.3.3.(2)+ Suppose the operations of addition and subtraction in Exercise 28.2.2 are replaced by comparison operations. Determine the CC setting in each case.

28.3.4.(2)+ Suppose the data in Exercise 28.2.3 is used in a comparison operation. Determine the CC setting in each case.

28.3.5.(2)+ By considering possible operand lengths and addresses for decimal addition, subtraction, and comparison, give a rigorous rule for the conditions under which the two operands may safely overlap.

28.3.6.(2)+ Compare the two 4-byte constants 1234+ to 2345− in both zoned and packed decimal representations, using CP and CLC instructions each time. What are your results?

28.4. Decimal Multiplication

The rules for multiplication of decimal operands obey the usual rules of algebra concerning signs. The product of the first operand (the multiplicand) and the second operand (the multiplier) replaces the first operand, which must be long enough to contain the digits of the result. Remember that the product of a number N1 digits long and another number N2 digits long is at most N1+N2 digits long.

The following rules apply to decimal multiplication:

1. All digits and signs are verified; any invalid digits will cause a decimal data exception, and the Interruption Code will be set to 7.
2. To ensure that the product will fit into the first operand field without overflow, there must be as many leading zero *bytes* in the first operand as the number of bytes in the second operand. Violating this requirement causes a data exception.
   - The length of the first (multiplicand) operand must be greater than the length of the second (multiplier) operand.
3. The System z architecture imposes an additional condition: the length of the second operand must not exceed 8 bytes, or 15 digits. If this or the previous condition is not met, a specification exception will occur, and the Interruption Code will be set to 6.

   Rules 2 and 3 avoid a potential multiplication overflow that could corrupt the product (multiplicand) field if the overflow was detected late in the multiplication process. We can usually choose the order of the two operands so that the rules are satisfied, because multiplication is commutative (independent of operand order).
4. The sign of the result is determined from the rules of algebra, even if one or both operands is zero. Thus, (2+)×(0−) will generate a minus zero.

Because the CPU takes great care to check the validity of both operands, no multiplication overflow can occur. This saves time and avoids data damage, because the multiplication need not be partially executed only to discover that the result is invalid. As with other System z multiply instructions, the CC setting is unaffected.

To illustrate a decimal multiplication, suppose we wish to multiply 126+ and 213+ (the operands we used in describing binary multiplication in Section 18.4):

```
  0000213+    (operand 1) (multiplicand, with 2 high-order zero bytes)
×     126+    (operand 2) (multiplier, 2 bytes long)
  0026838+    (product)
```

In this example, the operand lengths were chosen to have the minimum number of bytes needed to contain the given quantities, and still satisfy the above rules. However, the product still contains an "excess" high-order byte of zeros; this is a typical result in decimal multiplication. Multiplying 005+ and 5+ generates the result 025+, and shows that at least one leading zero digit is always found in the product. (See Exercise 28.4.2.)

> **Be Careful!**
>
> The results of packed decimal multiplication may depend on the order of the operands.

Exercises

28.4.1.(2) In decimal multiplication rule 2, why require the number of leading zero bytes in the first operand to be at least as many bytes as in the second operand?

28.4.2.(3) Show that the requirement that the number of leading zero bytes in a decimal multiplicand be no less than the number of bytes in the multiplier leads to at least one leading zero digit in the product.

28.4.3.(1) Give a reason why the System z architecture requires the second operand of a decimal multiplication to be no more than 8 bytes long.

28.4.4.(1) What is the largest packed decimal number that can be generated by a single multiplication?

28.4.5.(3)+ Determine the products of each of the following pairs of decimal numbers, assuming that the first number is the multiplier. Then do the same, assuming that the second number is the multiplier. Add enough leading zeros to each multiplicand so that the product will be valid.

(a) (9+) × (9−)
(b) (72−) × (7+)
(c) (44+) × (44+)
(d) (15−) × (55+)
(e) (107+) × (107+)
(f) (28+) × (3−)

28.4.6.(2)+ What result will be generated if you multiply (007+) and (009−)?

28.5. Decimal Division

Dividing two decimal numbers is more complicated than multiplying, since the result field must contain both the quotient and the remainder. The dividend (the first operand) is divided by the divisor (the second operand); the quotient and remainder then replace the dividend.

The method used is essentially the same as our familiar process of long division: the divisor is subtracted from the leftmost portion of the dividend, and the number of successful subtractions becomes the first quotient digit. The divisor is then shifted one digit position to the right, and the subtractions continue. This process ends when the rightmost digit of the divisor is aligned with the rightmost digit of the dividend, and no further subtractions can be performed. It is therefore natural that the remainder appears at the *right* end of the dividend (first operand) field, and the remainder has the same length as the divisor. The quotient is placed in the leftmost portion of the dividend field.[181]

Figure 288 illustrates how the operands and results appear in packed decimal division.

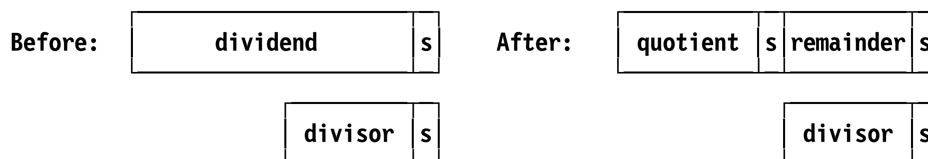


Figure 288. Operands for packed decimal division

Suppose 027+ (the dividend) is divided by 4+ (the divisor); the result that replaces the first operand is 6+3+. That is, the quotient is 6+ and the remainder is 3+. Note that both operands are signed; this means that there must be enough space in the first operand field for the quotient and remainder digits, and two sign digits.

Now, suppose 00787− is divided by 094+. We see that the result will be 8−035−; the quotient is 8− and the remainder is 035−. If we invert the signs of the dividend and the divisor and divide 00787+ by 094−, the result is 8−035+.

The rules for decimal division may be summarized as follows:

1. The sign of the quotient is determined from the usual rules of algebra, and the remainder sign always agrees with the dividend sign. (These rules apply even when the quotient or remainder is zero.)

[181] This is different from the process described for binary division in Section 18.8, where we visualized the process as shifting the dividend "across" the divisor; here, the divisor is shifted "under" the dividend.

2. The operands in a successful division always satisfy the relation

```
dividend = (quotient × divisor) + remainder,
```

   and the magnitude of the remainder is less than the magnitude of the divisor.
3. The length of the divisor is the length of the remainder, which is found at the right end of the first operand field.
4. The length of the divisor must be less than the length of the dividend so that there will be enough room for both quotient and remainder. Also, the System z architecture requires that the length of the divisor be 8 bytes or less. If either of these conditions is not met, a specification exception occurs.
5. If the divisor is zero, or if the quotient is too large to fit into the available space, the operation is *suppressed*: the dividend in memory remains unchanged, and a decimal divide interruption always occurs. The Interruption Code is set to 11.
6. Decimal overflow cannot occur, and the Condition Code remains unchanged.

Here is a simple test to determine whether a divide exception will occur: align the divisor and dividend so that the leftmost divisor digit is under the next-to-leftmost dividend digit. If the newly-positioned divisor is now less than or equal to the dividend, the division is improper and a decimal divide interruption will occur. For example, in dividing 00787+ by 094+, if we align the two operands as shown,

```
00787+    (dividend)
 094+     (divisor)
```

we see by comparing leading digits that 09 is greater than 007, so the division is proper. Conversely, if we divide 123456789+ by 987+, the alignment

```
123456789+    (dividend)
 987+         (divisor)
```

shows that the division is improper, since 9 is less than 12. Thus, the dividend *must* contain at least one leading zero digit.

The next two topics will concern some fairly technical aspects of packed decimal addition and subtraction on a binary processor. You may safely skip them if you are satisfied that the processor gets its results “somehow”, but the methods don't interest you.

Exercises

28.5.1.(2)+ Give examples which show how to generate a negative zero (1) as a sum, (2) as a product, (3) as a quotient, and (4) as a remainder.

28.5.2.(3) Determine the quotient and remainder for each of the following dividend-divisor pairs, and state any error conditions that might arise.

(1) (0009999+) ÷ (088+)
(2) (09999+) ÷ (099+)
(3) (0271828182845+) ÷ (4−)
(4) (0192519200513+) ÷ (370+)
(5) (012345678+) ÷ (321−)

28.5.3.(2)+ Explain why the result of dividing 00787− by 094+ cannot be 08−35−, rather than 8−035−.

28.6. True Decimal Addition (*)

The term “true addition” means the addition of two operands of *like* sign; we will discuss “complement addition”, the addition of two operands of *unlike* sign, in Section 28.7. True addition is simpler because the sign of the result is known in advance.

We will omit sign digits in the following examples, and assume that the sign digit of the first operand is left in memory as the sign of the final result.

Suppose we add two decimal digits using the four-bit binary representation for each decimal digit. A sum such as

```
  2       0010
+ 5     + 0101
  7       0111
```

gives the desired result both in decimal and in binary. However, if we add two digits whose sum exceeds 9, the decimal sum is correct, but the binary sum B'1100'= X 'C' is not a valid packed decimal digit:

```
  5       0101
+ 7     + 0111
 12       1100
```

Because we are using four bits to represent a decimal digit, the six bit combinations X'A'-X'F' must remain unused. A simple solution to this difficulty is to add 6 to the binary sum whenever it exceeds 9. In the above example, this “correction” process would be done as follows:

```
  0101      hex  5
+ 0111      hex  7
  1100      hex  C     = 12 decimal, > 9
+ 0110      hex  6     Correction
1 0010      hex 12
```

The two BCD digits in the result X'12' are the correct *packed decimal* representation of the sum, 12.

There is an obvious problem: how do we know *when* to add 6? In the first example, it would have been wrong to add 6, because the sum of 2 and 5 did not exceed 9. Furthermore, it will be difficult for the CPU to compare each sum digit to 9 (to see if a correction is needed), since a comparison implies a subtraction. Instead of correcting after adding, we do the following:

1. Add the digits of the two operands, *and* a corrector digit 6, at each digit position.
2. Note which digit positions generate a carry; the carry should be propagated.
3. In the digit positions from which *no* carry occurred, correct the sum by subtracting 6 (that is, by adding 10 = B'1010', the two's complement of 6) *without* propagating carries.

Here are some examples of this method. Let us add 2 and 5 again, including an extra high-order digit.

```
  02     0000 0010     first operand
  05     0000 0101     second operand
+ 66   + 0110 0110     corrector digits
  6D     0110 1101     intermediate sum, no carries occurred
+ AA   + 1010 1010     re-correction, add without carrying
  07     0000 0111     final sum
```

In this example, there were no carries out of either digit position in forming the intermediate sum, so that 6 had to be subtracted from both digit positions to arrive at the final sum.

Adding 95 and 87 (=182) would be done as follows:

```
  095     0000 1001 0101      first operand
  087     0000 1000 0111      second operand
+ 666   + 0110 0110 0110      add corrector digits
  782     0111 1000 0010      intermediate sum, two carries
              c    c
```

At this point, carries have occurred out of *both* of the low-order hex digit positions but not from the high-order digit; the final correction therefore involves a subtraction in only the leftmost digit:

```
  782     0111 1000 0010      intermediate sum
+ A00   + 1010 0000 0000      add complement of 6
  182     0001 1000 0010      final sum
```

Remember that carries are not propagated during the final re-correction step; otherwise, the sum would have been 1182 instead of 182.

Exercises

28.6.1.(3) Show by appropriate examples that true decimal addition can be performed as follows:

1. Perform a true binary addition, propagating carries.
2. In those digit positions from which *no* carry occurred, add 6, propagating carries.
3. In those digit positions from which no carry occurred in the second addition, subtract 6.

28.6.2.(2) Using the rules for true decimal addition given in this Section, perform the following sums, showing all the intermediate steps:

(1) (007) + (007)
(2) (007) + (009)
(3) (00987) + (00789)
(4) (00885) + (00595)

Signs were omitted because true addition involves like signs.

28.7. Complement Decimal Addition (*)

Complement addition of two decimal operands is more complicated than true addition, since the result may be of either sign. We will review the rules for complement addition first, and then give some examples.

1. The sign of the first operand is saved as the tentative sign of the result. The shorter operand is extended internally with high-order zeros to the length of the longer operand.
2. The numeric digit portion of the second operand (that is, everything but the sign digit) is complemented: this is the usual two's complement, obtained by inverting each bit, and adding a 1-bit in the low-order position.
3. The two operands are added. Carries are propagated from each 4-bit digit to the next, and whether or not a carry occurred is noted for each digit position.
4. To correct the result of the addition in step 3, subtract 6 (that is, add 10 = `B'1010'`, the two's complement of 6), in each digit position where *no* carry occurred in step 3. Carries out of a 4-bit group are not propagated to the next digit position during this decimal correction process.
5. If there was a carry out of the *high-order* digit position in step 3, the result is now complete. (We say that the result is in *true* form.)
6. If there was *no* carry out of the high-order digit position in step 3, the result is said to be in *complement* form. To obtain a correct result, invert the sign of the first operand (it was saved at step 1).

7. Form the two's complement of the result by inverting each bit and adding a 1-bit in the low-order digit position. In propagating the carries from this added 1-bit, note which 4-bit groups produce a carry and which do not.
8. This result is decimally corrected by subtracting 6 (adding binary 1010) to those digit positions from which *no* carry occurred during step 7. Carries generated in this correction process are not propagated.

We will now look at some examples of complement addition in which the result after step 5 is in true form; cases requiring recomplementation will be examined shortly. First, suppose we add 5+ and 2−. After saving the plus sign for the result, we proceed as follows:

```
  5     0101      first operand
+ E   + 1110      2's complement of second operand
c 3   c 0011      sum, with carry indicated by "c"
```

Since a carry occurred out of each digit position, no decimal correction is required; and since a carry occurred out of the high-order digit position, the result is in true form. Thus the final result is 3+, as expected.

To give an example in which a decimal correction (but no recomplementation) is needed, suppose we add 043+ and 019−. Saving the + sign for the result, we first form the two's complement of the second operand:

```
  1111 1110 0110      invert all bits of second operand (019)
+              1      add 1 in low-order position
  1111 1110 0111      two's complement of second operand
```

Now, we add this to the first operand:

```
  043     0000 0100 0011      first operand
+ FE7   + 1111 1110 0111      2's complement of second operand
  02A     0000 0010 1010      uncorrected sum, high-order carry
            c    c            carries
```

Since no carry occurred out of the rightmost hex digit position, we must decimally correct by subtracting 6 (adding binary 1010), without carrying:

```
  02A     0000 0010 1010      uncorrected sum
+ 00A   + 0000 0000 1010      add correction, no propagation
  024     0000 0010 0100      final true sum
```

The final sum is 024+, as expected.

To illustrate complement decimal additions in which a final recomplementation is required, we use the same two examples, but reverse the order of the operands. Thus, suppose we start by adding 2− and 5+. The tentative sign of the result is −, the sign of the first operand. We then proceed as before:

```
  2     0010      first operand
+ B   + 1011      2's complement of second operand
  D     1101      uncorrected sum, no carry
```

Because no carry occurred out of any digit position, we must add a decimal correction:

```
  D     1101      uncorrected sum
+ A   + 1010      decimal correction
  7     0111      sum, in complement form
```

The carry generated during the correction process is ignored. The result is known to be in complement form because there was no carry from the high-order (and only) digit position during the first addition. Thus, we invert the minus sign being saved for the result, and set it to a plus sign. To perform the final recomplementation, we first form the two's complement of the above result:

```
   1000      invert each bit of result
+     1      add 1 in low-order position
   1001      uncorrected 2's complement of result
```

Now, to obtain the final result, we observe that there was no carry out of the digit during the complementation process; we therefore add −6 to obtain the final sum:

```
  9    1001      uncorrected recomplemented result
+ A  + 1010      decimal correction
  3    0011      final corrected result
```

As noted, the carry generated during the final decimal correction is ignored, so the result is 3+, as expected.

As our final example, we add 019− and 043+. After saving the minus sign for the result, we take the two's complement of the second operand:

```
  1111 1011 1100      invert all bits
+              1      add a low-order 1-bit
  1111 1011 1101      2's complement of second operand
```

Now, we can do the first addition:

```
  019      0000 0001 1001      first operand
+ FBD    + 1111 1011 1101      2's complement of second operand
  FD6      1111 1101 0110      initial sum
```

Since there was no carry from the high-order digit position, we know the result is in complement form, and we invert the sign of the result to a plus sign. A carry occurred only out of the right-most digit position, so we must add a decimal correction to the two leftmost digit positions:

```
  FD6      1111 1101 0110      uncorrected initial sum
+ AA0    + 1010 1010 0000      decimal correction digits
  976      1001 0111 0110      sum in complement form
```

This is the result in complement form. To obtain the final result, we take the two's complement, and correct it by adding −6 in any position where no carry occurred during complementation:

```
           0110 1000 1001      bit-wise complement of 976
         +              1      add low order 1-bit
  68A      0110 1000 1010      complemented uncorrected result: no carries
+ AAA    + 1010 1010 1010      decimal correction digits
  024      0000 0010 0100      final result
```

As expected, the final result is 024+.

Exercises

28.7.1.(2) The three “signs” in an add or subtract operation are (1) the operation (+) or (−), (2) the first operand sign, and (3) the second operand sign. Make a table of the eight possible combinations of signs and verify that an even number of minus signs indicates a true addition, and an odd number of minus signs indicates a complement addition.

28.7.2.(5) Decimal numbers are sometimes represented in binary machines in the “excess-3” representation, whereby each decimal digit is represented by a hex digit whose value is larger by 3. For example, the number 280+ would be represented by 5B3+. Give rules like those in Sections 28.6 and 28.7 for true and complement addition in the excess-3 representation.

28.7.3.(3) Using the rules for complement decimal addition given above, perform each of the following operations.

(1) (004+) + (004−)
(2) (004−) + (004+)
(3) (0391−) + (1715+)
(4) (4837+) − (5537+)

28.7.4.(2) Assuming that recomplementation takes extra time, how could a list of decimal numbers of mixed signs be ordered to reduce the number of recomplementations and the time needed to compute their sum?

28.7.5.(3) In decimal complement addition, a result in complement form must be recomplemented and decimally corrected. During recomplementation, carries may or may not occur out of a 4-bit group. Show that in the low-order positions, a carry may occur only out of zero digits.

28.7.6.(2) Give examples of the possible combinations of digits to show that the addition scheme described above gives a correct decimal sum.

28.7.7.(3) Show by appropriate examples that the final correction of the sum (by subtracting 6 in certain digit positions) cannot cause a "borrow" from the next high-order digit.

28.7.8.(3) Examine the rules for true and complement decimal addition, and determine a general rule for the conditions under which carries are and are not propagated during the various intermediate addition steps.

28.7.9.(3) Determine the maximum number of additions required to add two one-byte decimal operands of any sign.

Terms and Definitions

complement decimal addition
The addition of packed decimal operands of unlike sign.

decimal divide exception
An exception condition caused by a packed decimal quotient being too large for the available space in the first operand field. A Program Interruption occurs, with Interruption Code `X'000B'`.

decimal overflow exception
An exception condition caused by a packed decimal sum or difference being too large for the receiving first operand field. The Condition Code is set to 3. If the Program Mask bit is set to 1, a Program Interruption occurs with Exception Code `X'000A'`.

decimal data exception
An exception condition caused by invalid numeric or sign digits in a packed decimal operand, or by a packed decimal multiplication or division specifying incorrect lengths for one or both operands. A program Interruption occurs with Exception Code `X'0007'` and Data Exception Code (DXC) `X'00'`.

non-overflowed zero
The addition or subtraction of packed decimal operands that generates a zero result always assigns a + sign

order dependence
Results of a packed decimal operation can depend on the order in which the operands are specified.

overflowed zero
The addition or subtraction of packed decimal operands that overflows can generate a zero result with a − sign

true decimal addition
The addition of packed decimal operands of like sign.

29. Packed Decimal Instructions

```
 2222222222    9999999999
222222222222  999999999999
22        22  99        99
          22  99        99
          22  99        99
         22   999999999999
       22     999999999999
     22                 99
   22                   99
 22           99        99
222222222222  999999999999
222222222222   9999999999
```

Table 209 lists the decimal arithmetic instructions in this section. MVO has rather specialized uses, and is not used often since SRP was introduced. Except for TP, each instruction has the format of a two-length SS-type instruction, as shown in Table 200 on page 469.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
FA	AP	SS	Add Decimal	FB	SP	SS	Subtract Decimal
FC	MP	SS	Multiply Decimal	FD	DP	SS	Divide Decimal
F9	CP	SS	Compare Decimal	F8	ZAP	SS	Zero and Add Decimal
F0	SRP	SS	Shift and Round Decimal	F1	MVO	SS	Move with Offset
EBC0	TP	RSL	Test Decimal				

Table 209. Packed decimal arithmetic instructions

The previous section sketched how the CPU does packed decimal addition, subtraction, multiplication, division, and comparison. Now, we'll examine the corresponding instructions.

All decimal operations process their operands from *right* to *left*. The operands should not overlap, or (in some cases) their rightmost bytes should coincide. If the operands overlap in any way, the CPU will store the result byte from one step of an operation before fetching the operand bytes to be used in the next step, or it will fetch the byte(s) for the next step before storing the result byte — in such cases, be sure to consult the *z/Architecture Principles of Operation* first.

Note!

All packed decimal arithmetic is *integer* arithmetic. Section 29.10 shows how you can do packed decimal arithmetic with mixed integer-fraction data (like 123.456×4.07).

29.1. TP Instruction

The TP instruction checks its packed decimal operand for validity. The Assembler Language syntax of a TP instruction is shown in Figure 289.

```
         TP    D1(N,B1)
```

Figure 289. Assembler Language syntax of the TP instruction

where the Length Expression N is the length of the packed decimal operand. The valid forms of the Assembler Language statement operand are shown in Table 210.

	Explicit Length	**Implied Length**
Explicit Address	$D_1(N,B_1)$	$D_1(,B_1)$
Implied Address	S(N)	S

Table 210. Operand formats for TP instruction

The format of the assembled machine instruction is illustrated in Table 211.

EB	L		B_1	D_1		C0

Table 211. Format of the TP instruction

You will remember from Sections 24.2 and 27.5 that the Encoded Length L is one less than the Length Expression N.

TP sets the Condition Code as shown in Table 212.

CC	**Meaning**
0	All digit codes and the sign code are valid.
1	The sign code is invalid.
2	At least one digit is invalid.
3	The sign code and at least one digit are invalid.

Table 212. CC settings for the TP instruction

Some of the examples and exercises in previous sections showed other ways to test the validity of a packed decimal operand. The TP instruction is generally much simpler; its only shortcoming is that it doesn't indicate which digit(s) or byte(s) may be invalid.

For example, we can test several operands:

```
         TP    T0                 Test operand T0; CC=0
         TP    T1                 Test operand T1; CC=1
         TP    T2                 Test operand T2; CC=2
         TP    T3                 Test operand T3; CC=3
         - - -
T0       DC    X'123456789D'      Valid operand
T1       DC    X'1234567890'      Invalid sign code
T2       DC    X'12345C789C'      Invalid digit
T3       DC    X'12345C7890'      Invalid sign and digit
```

Exercises

29.1.1.(2)+ For each of these operands, what will be the CC setting after testing it with a TP instruction?

(1) Z'330'
(2) P'-1945'
(3) X'567890'
(4) H'255'
(5) P'31415926535897914142135326607977277059'
(6) C'good!'

29.1.2.(2) Is there a way to use TP to detect the first of several possible invalid digit codes in a packed decimal operand?

29.1.3.(2) Suppose you know that a packed decimal operand has an invalid digit in the byte at **BadByte**. Write a short instruction sequence to test whether the left or right digit was bad, and branch to **BadLeft** or **BadRight** accordingly.

29.2. ZAP Instruction

The ZAP instruction[182] moves decimal data from the second operand to the first operand field. Its Assembler Language and machine instruction formats are the same as for PACK and UNPK, described in Section 27.

As the name implies, ZAP can be considered as equivalent to (1) setting the first operand field to 0+, and (2) adding the second operand to the newly-zeroed first operand.[183] That is, only the second operand must be a valid packed decimal number.[184] The first operand field may actually contain anything, and is not checked for validity.

- If the second (source) operand is exhausted before the first operand field has been filled, the CPU supplies extra high-order zeros until the first operand is complete.
- If significant (nonzero) high-order digits are lost because the first operand field is too short, a decimal overflow exception is recognized.
- If you use ZAP to initialize a field, sign codes of the second operand are converted in the first operand to the preferred signs, X'C' and X'D'.
- The sign of a zero result is +, unless there is an overflow; in that case, the zero result has the sign of the second operand.

ZAP isn't fussy about operand overlap, so long as the rightmost byte of the first (target) operand is not at a lower address than the rightmost byte of the second (source) operand.

ZAP sets the Condition Code as shown in Table 213.

CC	Indication
0	Result is zero.
1	Result is less than zero.
2	Result is greater than zero.
3	Decimal overflow.

Table 213. CC settings by the ZAP, AP, and SP instructions

Remember that a potential interruption for a decimal overflow condition can be masked off using the SPM instruction described in Section 16.9 on page 234.

182 The *Principles of Operation* calls it just “Zero and Add”, rather than “Zero and Add Decimal”. The “P” in the mnemonic helps to classify ZAP with other packed decimal instructions.

183 The name “Zero and Add” is slightly confusing, because the instruction does not *actually* zero the first operand and then add the second to it. Initially subtracting the first operand from itself might fail if it's not a valid packed decimal number!

184 Anything else will cause a data exception interruption.

The following statements show the effect of executing several ZAP instructions, where we use literals for the second (source) operands.

```
                         2nd Operand    Result
       ZAP   X,=PL1'1'   X'1C'          c(X) = 001+, CC = 2
       ZAP   X,=P'-9'    X'9D'          c(X) = 009-, CC = 1
       ZAP   X,=P'1234'  X'01234C'      c(X) = 234+, CC = 3 (overflow!)
       ZAP   X,=P'0234'  X'00234C'      c(X) = 234+, CC = 2
       ZAP   X,=P'0'     X'0C'          c(X) = 000+, CC = 0
       ZAP   X,=X'1234'  X'1234'        Data Exception (invalid sign, '4')
       ZAP   X,=X'ABCD'  X'ABCD'        Data Exception (invalid digit, 'C')
       - - -
X      DS    PL2                        First operand, two bytes long
```

Figure 290. Examples of the ZAP instruction

- The third ZAP instruction illustrates a decimal overflow due to the loss of significant high-order digits. The digits lost in the fourth statement are both zero, so no overflow occurs.
- The sixth ZAP instruction fails because the final digit 4 is not a valid sign, one of A-F.
- The last ZAP instruction fails because the first three digits are not numeric, one of 0-9; because data is accessed from right to left, the invalid digit X'C' in the first byte (X'CD') causes the decimal data exception.

If the first and second operand fields have the same length, it may be simpler to use an MVC instruction, assuming the second operand is valid.

To give some examples of the use of ZAP, suppose we must initialize a table of 50 three-byte packed decimal numbers starting at **Dec**. Each number in the table is three bytes long, and we must set them to zero. We'll use the “Branch on Count” (BCT) instruction (described in Section 22.4 on page 334) to control the loop:

```
NDec   Equ   50                   Number of elements
       LA    1,NDec               Initialize loop counter
       LA    2,Dec                Point to first element
Loop   ZAP   0(L'Dec,2),=P'0'     Zero an element
       AHI   2,L'Dec              Step pointer to next element
       BCT   1,Loop               Perform ZAP 'NDec' times in all
       - - -
Dec    DS    (NDec)PL3            Table of 3-byte elements
```

Figure 291. Using ZAP to initialize a table of packed decimal operands

where the ZAP instruction first operand has D_1=0, N_1=3 (so L_1=2), and B_1=2.

In this example, every number in the table has the same length and the entire table is only 150 bytes long, so we could also use an MVC instruction instead of executing a loop:

```
NDec    Equ   50                   Number of table elements
        ZAP   Dec,=P'0'            Initialize the first element
        MVC   Dec+L'Dec((NDec-1)*L'Dec),Dec    Propagate the zero
        - - -
Dec     DS    (NDec)PL3
```

Figure 292. Initializing a table of decimal numbers using MVC

where the ZAP instruction uses an implied length, the length attribute of **Dec**. If the length of the elements starting at **Dec** changes, we won't need to update the MVC instruction. Note also that N=(NDec−1)*L'Dec is the number of bytes at **Dec** *after* the first element.

This technique may be limited by the size of the table, and can't be used if the elements have different lengths.

ZAP is often used to move a shorter operand into a longer field; for example, ZAP is typically used to set up the first operand (the multiplicand) for decimal multiplications.

Exercises

29.2.1.(1)+ In Figure 290 on page 500, show the contents of the second byte of each ZAP instruction.

29.2.2.(1)+ Under what circumstances would it be useful to execute this instruction statement?

```
ZAP   A,A
```

29.2.3.(2)+ Suppose **X** is the name of a packed decimal operand two bytes long with value zero. Determine the effect of executing the instruction

```
ZAP   X,Y
```

if **Y** is the name of each of the following DC operands:

(1) PL5'-123'
(2) CL2' *'
(3) H'13'
(4) P'0000753'
(5) X'F00000123C'
(6) X'000123'

29.2.4.(2) Write one or more instructions to invert the sign of the packed decimal value at **PackVal**.

29.2.5.(1) As suggested following Figure 290 on page 500, you could use MVC instead of ZAP to initialize a packed decimal operand. Can you think of any reasons to avoid this technique?

29.2.6.(2) In Figure 292 on page 500, what is the maximum value of **NDec** for which the program segment will work correctly?

29.2.7.(2)+ Write instructions to replace the packed decimal number at **SomeVal** by its absolute value (that is, force the sign to be +), ensuring that the sign is the preferred sign, X'C'.

29.2.8.(3)+ Draw a detailed flowchart that describes the digit-by-digit action of ZAP. Take care to check the validity of sign and digits, set the CC and final sign, detect length incompatibilities, possible overflows, and operand overlap. (You will appreciate the complexities of implementing the instruction!)

29.3. AP and SP Instructions

Adding and subtracting decimal numbers is straightforward. The sum or difference replaces the first operand, and the usual CC setting (shown in Table 213 on page 499) reflects the status of the result.

AP and SP have the same machine instruction formats as ZAP, PACK, and UNPK, shown in Table 200 on page 469, and the assembler instruction statement operand formats shown Table in 201, both on page 469.

Unlike binary addition in the general registers, the results of decimal addition may depend on the order of the operands if the first operand field is too small. For example, adding 9+ to 123+ gives 132+, but adding 123+ to 9+ creates a decimal overflow:

```
         AP    PD123,PD9          Result = 132+
         AP    PD9,=P'+123'       Result = 2+, with decimal overflow
         - - -
PD123    DC    P'+123'            Packed decimal 123+
PD9      DC    P'+9'              Packed decimal 9+
```

The same consideration applies to subtraction, particularly for operands with opposite signs.

To illustrate the AP and SP instructions, Figure 293 assumes that the *initial* contents of the first operand field at **XX** is 143+ for each instruction (the instructions are not executed in sequence!). The result of executing each instruction is shown in its comment field.

```
                        2nd Operand   Result
        AP    XX,XX         X'143C'       c(XX) = 286+, CC=2
        SP    XX,XX         X'143C'       c(XX) = 000+, CC=0
        SP    XX,XX+1(1)    X'3C'         c(XX) = 140+, CC=2
        AP    XX,=P'-555'   X'555D'       c(XX) = 412-, CC=1
        SP    XX,=P'-555'   X'555D'       c(XX) = 698+, CC=2
        AP    XX,=P'1136'   X'01136C'     c(XX) = 279+, CC=3 (overflow)
        SP    XX,=P'1136'   X'01136C'     c(XX) = 993-, CC=1
        - - -
XX      DC    PL2'143'            Initial contents for all cases
```

Figure 293. Examples of the AP and SP instructions

In the third example, the second operand is one byte long, and starts at the second byte of **XX**.

The operands of SP and AP may overlap only if their rightmost bytes coincide, as in the first, second, and third instructions above. SP is rarely used to clear a field to 0+ as in the second instruction above, because ZAP (or MVC) are simpler and usually faster.

Suppose the table of decimal numbers at **Dec** that we zeroed in Figure 291 on page 500 now contains 50 data values, and that we wish to add them and place the sum at **SumDec**.

```
NDec   Equ   50                 Number of table elements
       ZAP   SumDec,=P'0'       Initialize sum to zero
       LA    0,NDec             Counter for number of summations
       LA    1,Dec              Pointer to first table element
Loop   AP    SumDec,0(L'Dec,1)  Add an element to sum
       LA    1,L'Dec(,1)        Step pointer to next element
       JCT   0,Loop             Branch until loop is complete
       - - -
SumDec DS    PL4                Accumulated sum
Dec    DS    (NDec)PL3          Table of 3-byte numbers
```

Figure 294. Adding a table of 50 packed decimal numbers

where the second operand of the AP instruction has D_2=0, N_2=3 (so L_2=2), and B_2=1.

Because each table entry is at most 5 digits long, the space allocated for the sum at **SumDec** does not have to be more than 7 digits (4 bytes) long. If the sum area is more than 4 bytes long, extra time might be needed each time the AP instruction was executed. Unlike addition and subtraction in the general registers, the time required for decimal addition and subtraction depends on the length of the operands.

Be Careful!

The results of packed decimal addition and subtraction may depend on the order of the operands if there is any possibility of decimal overflow.

Exercises

29.3.1.(2) A devious programmer thought he could ensure his continued employment by writing obscure code, such as this:

```
         PACK  A,*(1)              Do a packing task
         PACK  B,UNPK(1)           Do another, too
         AP    A,B                 Add some things
UNPK     UNPK  C,A(3)              Unpack the things
         - - -
A        DS    CL2
B        DS    B
C        DS    XL4
```

First, what do these instructions place in the fields named **A**, **B**, and **C**? Then rewrite his instructions without any obscurities, to show that you could easily take over his job.

29.3.2.(3)+ Suppose the two-byte field at **XX** initially contains 364+, and that the instruction

```
         AP   XX,XX+1(1)
```

is executed repeatedly until a decimal overflow occurs. How many times will the AP be executed?

29.3.3.(1) Compare the use of SP and XC (with identical first and second operands) for setting a decimal field to zero.

29.3.4.(3) Repeat Exercise 29.3.2 assuming that the field at **XX** initially contains 365+.

29.3.5.(1) Can a decimal overflow be caused by subtracting one operand from another, if both have the same sign?

29.3.6.(3) In Figure 293 on page 502, the third instruction uses an SP instruction with overlapping fields to zero a *part* of a packed decimal number. Devise other ways to do this, making no restrictive assumptions about the length of the operand or the amount of overlap.

29.3.7.(2)+ For the operands at **A** and **B**, show the result of executing these two AP instructions, assuming the same initial values of the operands.

```
         AP    A,B          Case 1
         AP    B,A          Case 2
         - - -
A        DC    P'0'
B        DC    X'075F'
```

29.3.8.(2)+ An instructor claimed[185] that this information came from a program interruption. Explain why the claim is false.

- Interruption Code = X'000A', Instruction Length Code = 1

29.3.9.(2)+ In Figure 294 on page 502, is the length of the **SumDec** field sufficient to hold the sum without overflow?

29.4. CP Instruction

The Compare Decimal instruction CP has the same Assembler Language and machine instruction formats as AP and SP.

Comparing two decimal operands only sets the CC, as shown in Table 214 on page 504. Overflow cannot occur.

---

[185] On an examination, of course!

CC	Indication
0	Operands are equal.
1	First operand is low.
2	First operand is high.

Table 214. CC setting by the CP instruction

The CP instruction is illustrated in the following examples; the result of the comparison is shown in the comment field of each statement.

```
         CP    XX,XX              CC = 0    123+ = 123+
         CP    XX+1(1),XX         CC = 1      3+ < 123+
         CP    XX,YY              CC = 2    123+ > 70493-
         CP    YY,XX              CC = 1  70493- < 123+
         CP    XX+1(1),YY+2(1)    CC = 2      3+ > 3-
         CP    XX(1),YY(1)        Interruption, invalid data
         - - -
XX       DC    P'+123' (=X'123C')
YY       DC    P'-70493'
```

In the last CP instruction, only the first byte of each operand is accessed, and an interruption will occur because neither byte contains a valid sign in its rightmost digit position.

To illustrate the CP instruction, suppose we must add the items in the same table of 50 decimal numbers as in Figure 294 on page 502. Now, assume that the positive and negative terms must be added separately, and the sums of their *magnitudes* are to be stored at **SumPos** and **SumNeg** respectively.

```
NDec     Equ   50                 Number of table elements
         ZAP   SumPos,=P'0'       Clear positive sum box
         ZAP   SumNeg,=P'0'       And negative sum box
         LA    0,NDec             Initialize term counter
         LA    1,Dec              Point to start of table
Loop     CP    0(L'Dec,1),=P'0'   Check sign of an entry
         JH    Plus               Branch to add + term
         JE    Next               Skip if it's zero
         SP    SumNeg,0(L'Dec,1)  It's -, accumulate negative sum
         J     Next               And go count to next item
Plus     AP    SumPos,0(L'Dec,1)  Accumulate positive sum
Next     LA    1,L'Dec(,1)        Step item pointer to next
         JCT   0,Loop             Repeat for the next item
         - - -
SumPos   DS    PL4                Positive sum
SumNeg   DS    PL4                Magnitude of negative sum
Dec      DS    (NDec)PL3          Table of elements
```

Figure 295. Adding positive and negative items separately

This program segment is straightforward; the JE instruction might save a small amount of time if zero elements are likely to appear in the table. The SP accumulates the magnitudes of the negative terms by subtracting them from **SumNeg**.

As another example, suppose we want to scan the table to find which element is algebraically largest, and leave its address at **BigItemA**.

```
         LA    0,NDec            Initialize count
         LA    1,Dec             And table pointer
Compare  CP    Max,0(L'Dec,1)    Compare current max to element
         JH    Next              Branch if max is bigger
         ZAP   Max,0(L'Dec,1)    Save element as new max value
         ST    1,BigItemA        And save address of that element
Next     LA    1,L'Dec(,1)       Step pointer to next element
         JCT   0,Compare         Count and loop
         - - -
BigItemA DS    A                 Address of largest element
Max      DC    PL3'-99999'       Initial maximum is a minimum
Dec      DS    (NDec)PL3         Table of elements
```

Figure 296. Finding the largest item in a table

Using JH in the fourth instruction rather than JNL may cause extra work to be done, but it guards against the remote possibility that all the table elements are equal to 99999−.

Exercises

29.4.1.(1)+ Give two reasons why it might be useful to compare a decimal number to itself.

29.4.2.(2) Suppose the table in Figure 296 contains more than one occurrence of the maximum value. Which one will the program find?

29.4.3.(2)+ Suppose you compare these two packed decimal operands using a CP instruction:

```
         CP    PDP,PDM           Compare operands
         - - -
PDP      DC    P'+0'
PDM      DC    P'-0'
```

What will be the resulting CC setting? Now, compare the same operands using a CLC instruction: what will be the resulting CC setting? How do the results depend on the sign codes of the operands?

29.4.4.(3)+ Rewrite the instructions in Figure 296 to eliminate all uses of a field (like the one named **Max**) in which the current maximum value is stored. Instead, set GR2 to the address of the largest element.

29.4.5.(2)+ What will be the Condition Code after executing these two instructions?

```
         CP    =P'+10',=P'-20'
         CLC   =P'+10',=P'-20'
```

Explain your results.

29.4.6.(2)+ In Figure 296, can the ZAP instruction cause a data exception? Would there be any difference if you replaced the ZAP instruction with

```
         MVC   Max,0(1)      ?
```

29.4.7.(1) In the last CP example following Table 214 on page 504, what are the values of the two operands being compared?

29.5. MP Instruction

Two length digits are provided in the MP instruction. The first digit specifies the length of the first (multiplicand) operand *and* of the product; the second digit specifies the length of the second (multiplier) operand. The two Encoded (machine) Length digits must satisfy the relations

$$1 \le L_1 \le 15, \quad 0 \le L_2 \le 7, \quad L_1 > L_2$$

and their Length Expressions must statisfy

$$2 \le N_1 \le 16, \quad 1 \le N_2 \le 8, \quad N_1 > N_2$$

(Section 24.1 describing basic SS-type instructions on page 365 explains the reasons for the differences between N, N_1, N_2, and L, L_1, L_2: the "L" values the CPU sees are one less than the "N" values you specify.)

It is important to remember that the number of bytes of high-order zeros in the *first* operand must not be less than the length of the *second* operand, even though the first operand may already contain enough high-order zeros.

Unlike binary multiplication in the general registers, these length restrictions mean that the results of decimal multiplication depend on the order of the operands. For example, multiplying 0000123+ by 456+ gives 0056088+, while multiplying 456+ by 0000123+ creates a specification exception, because the product field is too short to contain the result.

```
         MP    PD123,PD456        Result = 0056088+
         MP    PD456,PD123        Specification exception
         - - -
PD123    DC    PL4'+123'          4-byte packed decimal operand
PD456    DC    PL2'+456'          2-byte packed decimal operand
```

The first (multiplicand) operand field for a Multiply Decimal instruction is usually initialized with a ZAP instruction. This automatically sets the high-order digits of the first operand to zero. Thus, we could multiply 213+ by 126+ (as in Sections 18.4 and 28.4) as follows:

```
         ZAP   Prod,=P'213'       Set up multiplicand
         MP    Prod,=P'126'       Form product
         - - -
Prod     DS    PL4                Space for product
```

Figure 297. Example of decimal multiplication

The CC is unchanged by the MP instruction; in this example it will have been set to 2 by the preceding ZAP instruction to indicate the sign of the multiplicand.

To give a more elaborate example, suppose we wish to compute the square of each element of the table of fifty 5-digit numbers at **Dec** and store them in the table starting at **DecSq**. Each element of the table of products must be 6 bytes long, because there must be as many bytes of high-order zeros in the multiplicand as there are bytes (3) in the multiplier.

```
NDec     Equ   50                Number of table elements
LD       Equ   3                 Length of input table elements
         LA    0,NDec            Initialize counter
         LA    1,Dec             Point to input table
         LA    2,DecSq           Point to output table
Set      ZAP   0(2*LD,2),0(LD,1) Set up multiplier
         MP    0(2*LD,2),0(LD,1) Form (2*LD)-byte product
         LA    1,LD(,1)          Step 'Dec' table pointer
         LA    2,2*LD(,2)        Step 'DecSq' table pointer
         JCT   0,Set             Count down and loop
         - - -
Dec      DS    (NDec)PL(LD)      Table of input data
DecSq    DS    (NDec)PL(2*LD)    Squares of input elements
```

Figure 298. Using MP to square a table of decimal numbers

In this example, the number of table elements and the length of each element may be varied by modifying the EQU statements.

The MP instruction is easy to use, but there are times when some of its restrictions are annoying. For example, suppose we want to multiply the 4-byte operand 0001234+ by the 2-byte multiplier 101+; the result (0124634+) will fit quite comfortably in the four bytes provided by the original operand. However, because there are not as many bytes of high-order zeros as there are bytes in the multiplier, we must resort to schemes like this:

```
         ZAP   R,MPCand          Move multiplicand
         MP    R,=P'101'         Multiply by 101
          - -
MPCand   DC    PL4'0001234'      4-byte multiplicand
R        DS    PL5               Result must be 1 byte longer
```

Figure 299. Using ZAP to set correct decimal multiplicand length

The result at **R** will be 000124634+, but the extra byte of high-order zeros might not be needed for whatever operations are performed next.

Suppose as before that we want to multiply 0001234+ by 101+, but this time we reverse the order of the operands.

```
         ZAP   R,=P'101'         Lengthen the multiplicand
         MP    R,MPlier          Now multiply by 0001234+
         - - -
R        DS    PL6               Result at least 6 bytes long
MPlier   DC    PL4'1234'         4-byte multiplier
```

Figure 300. Using ZAP to set correct decimal multiplicand length

Because the multiplier is four bytes long, there must be *four* bytes of high-order zeros at R after the multiplicand 101+ has been placed there. Thus the result field must now be *six* bytes long, instead of the five required in Figure 299. Situations may arise where the order of the operands in a decimal multiplication is important![186] Forgetting to allocate the necessary extra bytes for the product will result in a data exception: a program interruption will occur, and the IC will be set to 7.

A final (but rarely troublesome) feature of decimal multiplication is that a negative zero result can be generated as the product of positive and negative zero operands. This is unlikely to occur in practice, because decimal addition and subtraction place a + sign on zero results (so long as no overflows occur). The generation of a negative zero is illustrated in Figure 301 on page 508.

[186] Mathematically, this means that the MP instruction is not commutative.

```
         MP    X,Y                Generate a negative zero product
         - - -
X        DC    PL2'0'             Positive zero
Y        DC    P'-0'              Negative zero
```

Figure 301. Generating 0− using MP

The product is X'000D', or 000−.

Be Careful!

Successful decimal multiplication often depends on the order of the operands.

Exercises

29.5.1.(1)+ Determine the inequalities that must be satisfied by the two Length Expressions in an MP machine instruction statement.

29.5.2.(1) What is the maximum value that may be assigned to the symbol **LD** in Figure 298 on page 507?

29.5.3.(2)+ Show why we cannot directly multiply 0001234+ by 100+ using an MP instruction, even though the product would contain a high-order zero digit.

29.5.4.(2) What will happen if we write MP X,X ?

29.5.5.(4) Suppose the CPU performed decimal multiplication by fetching one operand byte at a time, and storing a result byte as soon as it is generated. (It doesn't operate that way!) Under what circumstances could the first and second operands of an MP instruction overlap?

29.5.6.(2)+ What result will appear at **A** after executing the MP instruction?

```
         MP    A,A+1(1)
         - - -
A        DC    PL2'7'
```

29.5.7.(2)+ What result will appear at **B** after executing the MP instruction?

```
         MP    B,B+3(1)
         - - -
B        DC    PL4'567'
```

29.5.8.(3) What result will appear at **C** after executing the MP instruction?

```
         MP    C,C+3(3)
         - - -
C        DC    PL6'84567'
```

29.5.9.(1)+ Suppose you want to multiply 2 by 3, and write

```
         MP    A,B
         - - -
A        DC    PL1'2'
B        DC    PL9'3'
```

What result will be at A?

29.6. DP Instruction

As noted in Section 28.5, the results of a decimal division are found in the first operand field. Unlike binary division in the general registers, however, the quotient of a decimal divide is found in the left, or high-order, portion of the result, and the remainder is found in the right, or low-order portion. Take care in specifying the lengths of the operands: the remainder has the same length as the divisor, so the length of the quotient is the difference between the dividend and divisor lengths.

```
(quotient length) = (dividend length) - (remainder length)
                  = (dividend length) - (divisor length)
```

To avoid a specification error, the Encoded Length digits in the second byte of the instruction must satisfy the same inequalities as for MP:

$$1 \le L_1 \le 15, \quad 0 \le L_2 \le 7, \quad L_1 > L_2$$

and similarly for the corresponding Length Expressions.

To illustrate, suppose we want to divide 162843+ by 762+ (as in Section 18.8) using packed decimal division, as shown in Figure 302.

```
         ZAP   Dvnd,=P'162843'    Initialize dividend
         DP    Dvnd,=P'762'       Divide by 762+
         - - -
Dvnd     DS    PL4                Quotient = 213+, remainder = 537+
```

Figure 302. Decimal division using DP

After the ZAP instruction is executed, the area named **Dvnd** will be initialized to 0162843+. Because the divisor is 762+, the alignment test described in Section 28.5 will be satisfied, and a valid quotient and remainder can be computed. After the DP instruction is executed, the four bytes at **Dvnd** will contain 213+537+, so the quotient is 213+ and the remainder is 537+.

It's often easier to let the Assembler determine operand lengths by using Symbol Length Attribute References, as shown in Figure 303.

```
         ZAP   Dvnd,=P'162843'    Initialize dividend
         DP    Dvnd,Dvsr          Divide by divisor
         ZAP   Quot,Dvnd(L'Dvnd-L'Dvsr)       Move quotient...
         ZAP   Rem,Dvnd+L'Dvnd-L'Dvsr(L'Dvsr) ...and remainder
         - - -
Dvnd     DS    PL4                Can be up to 16 bytes long
Dvsr     DC    P'762'             Divisor
Quot     DS    PL5                Space for quotient  000000213C
Rem      DS    PL4                Space for remainder 0000537C
```

Figure 303. Decimal division using Length Attribute References for operands

In this example, the areas named **Quot** and **Rem** were purposely chosen to be longer than required, to show that extra length makes no difference in the values that will be stored. So long as none of the restrictions on operand lengths are violated, this instruction sequence will give correct results.

As a final example, suppose we must calculate the average of the entries in a table of nonzero 9-digit packed decimal numbers stored beginning at **Tbl**, and that we don't know the length of the table, only that the last entry is followed by a zero element.

```
ELen     Equ   5                  Length of a table entry (9 digits)
         LA    1,Tbl              Point to origin of table
         ZAP   Sum,=P'0'          Initialize sum box
         ZAP   Nbr,=P'0'          Initialize element counter
         ZAP   Avg,=P'0'          And set average to zero
Test     CP    0(ELen,1),=P'0'    Check for table end
         JE    Divide             Found end, go compute average
AddUp    AP    Sum,0(ELen,1)      Accumulate sum
         AP    Nbr,=P'1'          And increment counter
         LA    1,ELen(,1)         Step pointer to next entry
         J     Test               And loop
Divide   CP    Nbr,=P'0'          Check for no data at all
         JE    Finish             Give up if no entries
         DP    Sum,Nbr            Perform division
         ZAP   Avg,Sum(L'Sum-L'Nbr)    Move result
Finish   - - -
         - - -
Sum      DS    PL15               Accumulated total of entries
Nbr      DS    PL4                Count of entries being summed
Avg      DS    PL(ELen)           Average value
Tbl      DS    5000PL(ELen)       Lots of room for data
```

Figure 304. Computing the average of a table of decimal numbers

Useful Rule of Thumb

The field into which you ZAP a dividend prior to a packed decimal division should be at least as long as the length of the dividend plus the length of the divisor. (The symbolic forms in Figure 303 on page 509 are helpful for valid divisions.)

Exercises

29.6.1.(2)+ Revise Figure 304 on the assumption that the end of the table is signaled by a zero element with a *negative* sign. Positive zeros are permitted among the table entries.

29.6.2.(1)+ Determine the inequalities that must be satisfied by the Length Expressions in a DP instruction statement.

29.6.3.(2)+ What will happen if we write DP X,X ?

29.6.4.(1) Give at least two methods for testing whether a packed decimal number is even or odd.

29.6.5.(2) In Exercise 29.6.1, why can we not test for the terminating negative zero element with this statement?

```
Test     ZAP   0(ELen,1),0(ELen,1)      Test for -0 element
         JM    NegZero                  Jump if -0
```

29.6.6.(2) It is possible to write DP and MP instruction statements that will specify L_1 digits, where L_1 has value zero. If this is done, what will happen when the instructions are executed?

29.6.7.(4) Suppose the CPU performs decimal division by fetching one operand byte at a time, and storing a result byte as soon as it is generated. (It doesn't operate that way!) Under what circumstances could the first and second operands of a DP instruction overlap?

29.6.8.(2)+ What result will appear at **A** after executing this DP instruction?

```
         DP    A,A+3(1)
         - - -
A        DC    PL4'67'
```

29.6.9.(2)+ What result will appear at **B** after executing this DP instruction?

```
         DP    B,B+3(2)
         - - -
B        DC    PL5'567'
```

29.6.10.(3) What result will appear at **C** after executing this DP instruction?

```
         DP    C,C+3(2)
         - - -
C        DC    PL5'9567'
```

29.6.11.(1) Can you think of a reason why the designers of System z required the length of the second operand of a decimal division to be 8 or fewer bytes?

29.6.12.(2)+ Explain why there must be at least one leading high-order zero digit in the dividend of a decimal division.

29.6.13.(3) In a division process, the remainder may be chosen in (at least) three ways: (1) the remainder has the same sign as the dividend, (2) the remainder has the same sign as the quotient, and (3) the remainder is always nonnegative. In all cases, the magnitude of the remainder is less than the magnitude of the divisor.

In System z, the first alternative is used. What modifications would have to be made to the rules for decimal division if the second or third alternatives had been chosen instead?

29.6.14.(2)+ What will be the result at A of the following division?

```
         DP    A(5),A+5(2)
         - - -
A        DC    XL7'3876543C047C'
```

29.7. SRP Instruction

The SRP instruction shifts its operand to multiply or divide a packed decimal number by a power of 10, and to round the quotient of such a power of 10 “division”.

Before SRP was available, shifting a *computed* number of digit positions was complicated: different coding techniques had to be used for shifting left or right by an even or odd number of digit positions.[187] It was also difficult to use the EX instruction to specify the number of digits to be shifted. The CPU architects rectified this unhappy situation by creating the SRP instruction, which shifts in either direction with optional rounding for right shifts.

The Assembler Language statement format for an SRP instruction statement is shown in Figure 305, where the first operand address is given by $D_1(B_1)$, the Effective Address of the second operand $D_2(B_2)$ specifies the number of digits to be shifted, and I_3 is the rounding digit.

$$\texttt{SRP}\quad D_1(N_1,B_1),D_2(B_2),I_3$$

Figure 305. Assembler Language format of SRP machine instruction statement

Table 215 shows the assembled machine instruction. format of the SRP instruction:

F0	L_1	I_3	B_1	D_1	B_2	D_2

Table 215. Format of the SRP instruction

[187] If you're interested, we'll see examples in Section 29.9.

The SRP instruction format has several unusual features.

1. There is a single 4-bit length field, L_1.
2. The immediate digit I_3 is used for rounding only when shifting to the right. It must be a valid decimal digit; if not, a decimal data exception occurs, and the IC is set to 7. (The value of the I_3 digit is *not* specified by nor inferred from the length of the second operand.)
3. The decimal number to be shifted is in memory at the Effective Address computed from the first addressing halfword.
4. The Effective Address computed from the second addressing halfword determines *both* the direction *and* the amount of the shift. It is *not* a memory address.

The L_1 digit and the first addressing halfword provide the usual method for referring to a packed decimal operand. The effective address computed from the second addressing halfword is evaluated by the CPU, and (unlike binary shifts) its rightmost six bits are treated as a *signed* two's complement integer. The number of digits to be shifted therefore lies in the range

$$\text{B'100000'} = -32 \leq \text{shift count} \leq +31 = \text{B'011111'}$$

The shift is to the left if the shift count is positive, and to the right if the shift count is negative. You can think of the shift count as the power of ten P by which the decimal operand will be multiplied. A zero shift count may be specified, but it only causes the decimal operand to be checked for validity, which is done at the start of every SRP operation.

If the shift is to the left, the CPU checks for nonzero high-order digits, and indicates a decimal overflow condition if any are lost. If the shift is to the right, the I_3 digit is added decimally to the last digit shifted out of the first operand, and any carry is propagated into the remaining part of the first operand. The I_3 digit is considered to have the same sign as the decimal number being shifted. For both left and right shifts, vacated digit positions are set to zero. The CC settings after SRP are the familiar values shown in Table 213 on page 499.

You can specify the second-operand shift amount using the rules for the two's complement representation: because the shift count is a number P between −32 and +31, its 6-bit two's complement representation is simply 2^6+P, or 64+P. Thus, a right shift of two places with no rounding is specified by this instruction statement:

```
         SRP   A,64-2,0            Shift right 2, no rounding
```

Similarly, a left shift of two places is specified by

```
         SRP   A,64+2,0            Shift left 2 digit positions
```

When shifting left, the value of the I_3 digit is ignored (but it still must be a valid decimal digit!). The extra factor of 64 in the shift count will be ignored when the rightmost 6 bits of the effective second operand address are used; including the factor of 64 ensures that we won't forget it if it's needed for right shifts.

Suppose we want to shift the operand 12345+ to the left three places, as in Figure 306.

```
         SRP   A,3,0               Left shift causes an overflow
         - - -
A        DC    P'12345'            Final contents = 45000+
```

Figure 306. Shifting a decimal operand left 3 places using SRP

The CC will be set to 3 to indicate the decimal overflow condition.

To shift the same operand to the right without rounding, we can use SRP as in Figure 307.

```
         SRP   A,64-2,0            Shift right 2 digits, no rounding
         - - -
A        DC    P'12345'            Final value = 00123+
```

Figure 307. Shifting a decimal operand right 2 places using SRP

To illustrate rounding, let's shift 12345+ to the right one digit position with I_3 rounding digit 9 (so that “rounding” will occur if the last digit shifted has any nonzero value). We set the rounding digit to 9 to show that it may be any value, not just the value 5 normally used for rounding.

```
         SRP   A,64-1,9           Shift right, rounding digit = 9
         - - -
A        DC    P'12345'           Final value = 01235+
```

Figure 308. Shifting a decimal operand right 1 place with rounding using SRP

An important property of SRP is that the rounding digit can be provided at execution time by an EX instruction. For example, suppose the rounding digit is contained in the one-byte packed decimal number at **RndDigit** and the shift amount is stored as a halfword binary integer at **ShiftAmt**. We can then shift the operand as shown in Figure 309, where the shift amount is contained in GR1, and the rounding digit in GR2 is ORed into the second byte of the SRP instruction in the Instruction Register as the target of EX.

```
         LH    1,ShiftAmt         Get shift amount
         XR    2,2                Clear GR2 for rounding digit
         IC    2,RndDigit         Get rounding digit and its sign
         SRL   2,4                Drop off sign code
         EX    2,SRP              Execute the SRP instruction
         - - -
SRP      SRP   A,0(1),*-*         Executed instruction
A        DC    PL16'59365'        Operand to be shifted
RndDigit DC    PL1'9'             Rounding digit, packed decimal form
ShiftAmt DC    H'-2'              Shift amount
```

Figure 309. Shifting a decimal operand with an EXecuted SRP

In Section 29.9 we will see examples showing why SRP is such a useful decimal instruction.

29.7.1. Biased and Unbiased Rounding with SRP (*)

The rounding provided by SRP is slightly biased. To see why, suppose a bank pays daily interest in cents, and the calculated amounts for four days are 1.5, 2.5, 3.5, and 4.5 cents. If these are rounded to single cents using SRP with rounding digit 5 and a single right shift, the amounts added to the account are 2, 3, 4, and 5 cents, or a total of 14 cents. But the true total of the unrounded amounts is 12 cents, so the bank would be overpaying the customer.[188]

We may need to provide unbiased rounding for packed decimal results. Consider this example with a single “rounding digit”, as in 337785+, where the 5 is the digit to be rounded. Using a single right shift and a SRP rounding digit 5, the biased result would be 33779+. The rule for unbiased rounding is:

> If the value to be rounded lies exactly halfway between two possible rounded values, choose the rounded value with an *even* low-order digit.

Thus, in this example, the unbiased rounded value is 33778+. The unbiased rounded daily interest values would be 2, 2, 4, and 4 cents (a total of 12 cents), and the bank would not be overpaying its customer.

These instructions show one way to do unbiased rounding of a packed decimal number with a single rounding digit:

[188] Banks don't like to overpay their customers. (Of course, a bank could truncate the fraction part and keep the accumulated “breakage”.)

```
         MVZ   RDigit(1),PDVal+L'PDVal-1    Save rounding digit
         CLI   RDigit,X'50'      Is the rounding digit exactly 5?
         JNE   RoundUp           No, do a normal round up
         TM    PDVal+L'PDVal-2,1  Is the 'to-be-rounded' digit odd?
         JO    RoundUp           If yes, round up to an even value
         SRP   PDVal(L'PDVal),64-1,0   No: round down to even
         J     Done              Rounding completed
RoundUp  SRL   PDVal(L'PDVal),64-1,5   Round up to even
Done     - - -
Rdigit   DC    X'00'             Single rounding digit tested here
PDVal    DC    P'337785'         Decimal value, single rounding digit
```

See Exercises 29.7.14 and 29.7.15 for general forms of unbiased rounding.

We'll see in Chapter IX that decimal floating-point provides many more forms of decimal rounding, and requires many fewer instructions.

Exercises

29.7.1.(1)+ A programmer who wanted an SRP instruction to shift an operand two digits to the right wrote the statement

```
        SRP   A,-2,0
```

and received a diagnostic message from the Assembler. Attempting to correct the statement, he wrote

```
        SRP   A,0-2,0
```

and received the same diagnostic. Why did he receive these diagnostic messages? What is wrong with these statements? What should be done to fix them?

29.7.2.(2)+ Prove that no overflow exception can occur when executing an SRP instruction when the shift is to the right.

29.7.3.(2)+ In an SRP instruction performing a right shift, why is the I_3 digit added *decimally* to the last digit shifted out, rather than just added?

29.7.4.(2) What will happen if you execute this SRP instruction on each of the given operands?

```
         SRP   A,64-1,5
         - - -
A        DC    X'123456'         First sample operand
A        DC    X'ABCDEF'         Second sample operand
```

29.7.5.(2) What will be the result at **X1**, **X2**, and **X3** after executing these SRP instructions?

```
         SRP   X1,0,5
X1       DC    P'998'

         SRP   X2,1,5
X2       DC    P'95'

         SRP   X3,64-1,5
X3       DC    P'95'
```

29.7.6.(1) What shift amount is specified by these instructions, and what resulting data will appear at **A**?

```
         LHI   2,-2              c(GR2) = -2
         SRP   A,1(2),3          Shift the data somehow
         - - -
A        DC    PL3'-4567'        Test data
```

29.7.7.(2) After executing this SRP instruction, what result will appear at **XX**?

```
         SRP   XX,63,+5           Shift data at XX
         - - -
XX       DC    PL3'-4567'         Sample data
```

29.7.8.(2) In Figure 309 on page 513, what result will appear at **A** and what will be in the rightmost 6 bits of GR1? Why is the shift amount in GR1 not reduced by 1, as is done for EXecuted instructions like MVC?

29.7.9.(5) A program contains

1. a packed decimal operand at **A** whose length is given by its length attribute L'A,
2. a positive one-byte decimal number at **RND** satisfying $0 \leq c(RND) \leq 9$, and
3. a halfword binary integer at **SHFT** satisfying $-32 \leq c(SHFT) \leq +31$.

Write a program segment (not using SRP) to simulate the action of SRP on the operand at A, as illustrated in Figure 309 on page 513.

29.7.10.(2) SRP and the general-register shift instructions discussed in Section 17 all use the low-order six bits of an Effective Address for the shift amount. Why do you think SRP treats this amount as a signed number, while the binary shifts treat it as an unsigned number?

29.7.11.(1) If you thought SRP meant "Shift Right Packed" but found that was incorrect, would you have any reasons to prefer that it be named SLRP, meaning "Shift Left or Right and Round Decimal"? Why might this actually be a better choice?

29.7.12.(2) A careful programmer wanted to be certain that his SRP instruction would correctly round a negative packed decimal operand, so he wrote

```
         SRP   NegData,64-1,-5    Round a negative operand correctly
         - - -
NegData  DC    PL4'-72945'        Negative operand to be rounded
```

What will this instruction do?

29.7.13.(2)+ What is the result at **E** of executing this SRP instruction?

```
         SRP   E,2,5
         - - -
E        DC    X'075F'
```

29.7.14.(5) When 5 is used as the rounding digit in an SRP instruction, the rounded result is slightly *biased*, because values exactly halfway between two possible rounded results are always forced to the larger magnitude.

Write instructions that will round packed decimal numbers D digits long to R digits (where $R < D$; that is, the number of digits to be discarded after rounding is $D-R$), in such a way that initial values lying exactly half way between two possible rounded R-digit results always give a result with an *even* low-order digit.

For example, if D=5 and R=3, two right shifts are required; if the source operand is 12550+ then the rounded result will be 00126+, but 12450+ will be rounded to 00124+, not to 00125+ as SRP would do.

29.7.15.(5) Suppose the packed decimal number at **PDVal** is B bytes long, and has R rounding digits, where $R < 2 \times B-1$. (There will be $2 \times B-1-R$ digits left after rounding.) Write instructions that will leave an unbiased rounded result at **PDVal**. For example, if there are R=3 rounding digits, and the number is 33778529+, then the rounded result should be 00033779+; but if the number is 33778500+, the rounded result should be 00033778+.

29.7.16.(2) It is claimed that a rounded quotient can be formed this way:

1. Shift the dividend left one digit.
2. Divide.
3. Shift the quotient right one place, with rounding digit 5.

Prove or disprove this claim.

29.7.17.(2) Show at least four ways to generate a packed decimal negative zero.

29.8. MVO Instruction

MVO simply moves data from the second operand field to the first, with no validity checking and no effect on the CC. Its Assembler Language syntax is

```
MVO   D1(N1,B1),D2(N2,B2)
```

MVO moves hex digits from the second operand field to the first operand field in a way similar to ZAP, with one very important difference: the sign digit of the first operand is left in place, and *all* the digits (including the sign) of the second operand are placed to the left of the first operand sign. Zeros are supplied in the high-order positions of the first operand if the second operand is exhausted.

This process is illustrated in Figure 310, where the digits of the second operand are named "a", "b", etc., and the sign digits of the first and second operands are named "s1" and "s2".

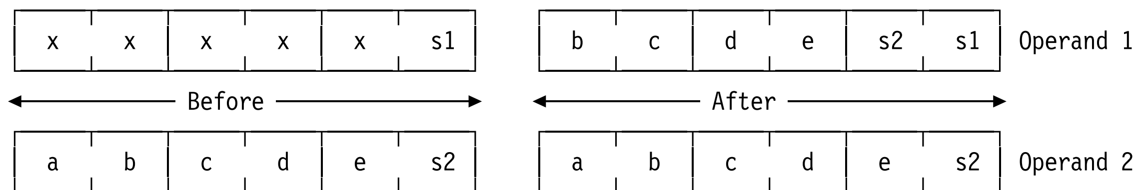


Figure 310. Operation of the MVO instruction

For most applications, the first and second operands will not overlap. We can visualize the sequence of operations performed by the CPU in executing an MVO instruction this way:

1. Move the second operand to an internal work area, and shift it left (*offset* it) by one hex digit.
2. Attach the sign digit of the first operand to the right end of this internal digit string, and place as many high-order zero digits at the left end as may be needed to fill all bytes of the first operand.
3. Move the resulting digit string to the first operand area, starting at the right-hand end. Excess high-order digits are lost.

Figure 311 illustrates using MVO with nonoverlapping operands.

```
         MVO   A,B                Move a shorter to a longer, and...
         MVO   C,D                Move a longer to a shorter.
         - - -
B        DC    P'678'             Two-byte operand
A        DC    PL4'-5'            Initial contents = X'0000005D',
*                                 Final contents = X'000678CD'
         - - -
D        DC    P'987654321'       Five-byte operand
C        DC    PL3'555'           Initial contents = X'00555C',
*                                 Final contents = X'4321CC'
```

Figure 311. Two Examples of MVO results

Figure 311 shows that the result of an MVO instruction can be an invalid decimal operand, because the rightmost byte may contain two sign codes instead of one.

This example does not indicate the contexts in which an MVO instruction might normally appear. The next section contains examples of its most common early application, decimal shifting. You can of course perform a decimal "shift" by multiplying or dividing by an appropriate power of ten, just as a binary shift multiplies or divides by a power of two. However, SRP is faster than the corresponding multiplication and division instructions.

While MVO is rarely used with overlapping operands, the result is as though bytes are processed one at a time with each result byte being stored immediately after fetching the source bytes.

Some examples showing how MVO can be used for shifting are described in Section 29.9.

Exercises

29.8.1.(4) Given that N_1 and N_2 are the true lengths of the first and second operands, write instruction sequences to *simulate* an MVO instruction assuming

1. $N_1 = N_2$,
2. $N_1 < N_2$, and
3. $N_1 > N_2$.

29.8.2.(1)+ What will be the result (in hexadecimal) at **New** of executing this instruction, for each of the source operands?

```
         MVO   New,Old
         - - -
New      DC    C'12345*'
```

```
 1. Old   DC  X'123456'
 2. Old   DC  C'ABCD'
 3. Old   DC  F'123456'
 4. Old   DC  X'123456789ABC'
```

29.8.3.(2)+ What can you say about the number of numeric digits moved by an MVO instruction?

29.8.4.(2)+ If an MVO instruction results in truncated second-operand digits, what can you say about the number of truncated digits?

29.8.5.(3)+ Draw a detailed flowchart that describes the digit-by-digit action of MVO. Be sure you correctly handle operands of unequal lengths, and the possibility of operand overlap. (Can you use the similarities between MVO and ZAP to combine this flowchart with the one you drew for ZAP in Exercise 29.2.8?)

29.8.6.(2)+ You have now encountered several instructions which can be used to move individual hex *digits* from byte to byte. Let the left and right hex digits of **Byte1** be labeled 1L and 1R, respectively, and similarly for digits 2L and 2R of **Byte2**. Now, make a table which shows all of the possible ways to move one or both digits from **Byte1** to **Byte2**, and an instruction that will perform the specified movement.

29.8.7.(3)+ A programmer suggested this technique for converting the bits of a byte to eight printable 0 and 1 EBCDIC characters:

```
         MVC   Out(1),Byte
         MVC   Out+1(7),Out
         NC    Out,Mask
         MVO   Work,Out(4)
         MVC   Out(4),Work
         TR    Out,=C'011111111'
Byte     DC    B'11010110'
Mask     DC    X'8040201008040201'
Work     DC    XL5'7777777777'
Out      DC    XL8'aaaaaaaaaaaaaaaa'
```

First, explain how it works. Then, explain why the literal is nine characters long.

29.8.8.(2)+ Assuming the lengths of the first and second operands of MVO are N_1 and N_2 respectively, (a) how many digits are lost if $N_1 \geq N_2$, and (b) how many high-order zero digits are supplied by the CPU if $N_2 < N_1$?

29.9. Decimal Shifting Using MVO (*)

When we need to multiply or divide packed decimal numbers by a power of ten, it's easiest to use the SRP instruction. Unfortunately, the original architecture of System/360 provided *no* basic instruction to shift decimal operands; you had to write an explicit instruction sequence for each shift. We'll examine some techniques that can be used to shift packed decimal operands “the hard way”.

You will appreciate the power of the SRP instruction by comparing it to what had to be done when it was not available.

29.9.1. Shift Right an Odd Number of Digits

To perform a decimal *right* shift of an odd number of digit positions, only the MVO instruction need be used. For example, to shift 12345+ to the right by three digit positions, we could write instructions as in Figure 312.

```
         MVO   A,A(1)             Shift right 3 decimal digits
         - - -
A        DC    P'12345'           Initial contents = 12345+
```

Figure 312. Shifting a decimal operand right an odd number of digits

Because the contents of the byte at **A(1)** is X'12', the final contents of the 3-byte memory area named **A** would be 00012+, as desired.

To illustrate this technique in greater generality, suppose an operand at **A** of length L bytes is to be shifted right N decimal digit positions, and we know N is odd and that $N < 2L-1$. The required MVO instruction can be written as in Figure 313:

```
         MVO   A(L),A(L-N/2+1)    Shift right N decimal digits
```

Figure 313. Shifting a decimal operand right an odd number of digits

Although the first and second operands overlap, it's easy to see how this instruction works, since the data is being shifted to the right: this is the “natural” direction for MVO.

29.9.2. Shift Left an Odd Number of Digits

To perform a decimal *left* shift of an odd number of decimal places, we need more than one instruction. If we want only to introduce an odd number of zeros at the right end of the number without dropping off any digits at the left, this simple technique can be used. First, we move the sign to the right N+1 digit positions, and then we move the digits themselves to the right one digit position. This effectively introduces N vacant digits, which can then be set to zeros. To illustrate, suppose the packed decimal operand 12345+ at **A** is to be shifted left 3 decimal digit positions and placed at **AShifted**.

```
         MVN   A+4(1),A+2         Move sign 4 digits to right
         MVO   A(4),A(3)          Shift digits right once
         NC    A+3(2),=X'000F'    Remove unwanted digits
         - - -
AShifted DS    0PL5               Result = 012345000+
A        DC    P'12345'           Initial operand = 12345+
         DS    PL2                Space for 3 inserted zeros and sign
```

Figure 314. Shifting a decimal operand left an odd number of digits

The symbol **AShifted** is defined in such a way that we can refer to the shifted (and extended) operand with the correct address and its new length attribute. (See Exercise 29.9.2.)

If the shifted operand must have the same length as the original operand, the code required is slightly more complicated, *unless* only a single digit shift is required. This case can be done simply, as illustrated in Figure 315. Suppose the initial operand is again 12345+.

```
         MVO   A,A                Shift left one decimal digit
         NI    A+L'A-1,X'0F'      Set duplicated sign to zero
         - - -
A        DC    P'12345'           Final contents = 23450+
```

Figure 315. Shifting a decimal operand left by one digit

In this case the operands overlap completely. The rule used by the CPU to handle possible memory conflicts is that any second operand bytes needed for any step of the operation are fetched from memory just before storing the result byte of that step.

If the first and second operands must have the same length for odd left shifts of 3 or more digits, then it is best to use a separate area for the first operand. To illustrate, suppose the packed decimal operand 12345+ must be shifted left 3 places, and the result is stored at **B**. As in Figure 306 on page 512, we'll shift the operand 12345+ to the left three places, as shown in Figure 316.

```
         MVN   B+2(1),A+2         First, move sign digit
         MVO   B(2),A             Move sign and two digits (45+x)
         NC    B+1(2),=X'000F'    Set sign and extra digits to zeros
         - - -
A        DC    P'12345'           Initial contents = 12345+
B        DS    PL3                Final contents = 45000+
```

Figure 316. Shifting a decimal operand left by three or more digits

This technique can of course be used for single left shifts, but the method of Figure 315 is simpler. (See Exercise 29.9.3.)

29.9.3. Shifting an Even Number of Digits

Shifting a decimal number an even number of digit positions is usually simpler than for the odd shifts, because the digits retain their relative positions within each byte. Thus, we usually do even shifts with the ordinary move and logical instructions. For example, to “multiply” 0001234+ by 100 (*not* by the packed decimal operand 100+), we need only shift the digits left by two positions, or one byte:

```
0001234+ × 100     becomes     0123400+.
```

This result can't be obtained by using an MP instruction, because there are not enough high-order zeros in the operand to be “shifted”.

29.9.4. Shifting Left an Even Number of Digits

To shift *left* using move and logical instructions, we can use the technique shown in Figure 317.

```
         MVC   A(3),A+1             c(A) = 01234+4+
         NI    A+3,X'0F'            c(A) = 01234+0+
         NI    A+2,X'F0'            c(A) = 0123400+, now 100 × greater
         - - -
A        DC    PL4'1234'            Initial data = 0001234+
```

Figure 317. Shifting a decimal operand left an even number of digits

In this example, the length of the operand is the same before and after the operation. If you need a longer result, you can use the technique shown in Figure 318: instead of shifting the data to the left, we shift the sign to the right.

```
         MVN   B+4(1),A+3           Move sign digit only
         NI    B+4,X'0F'            Zero out the excess digit
         NI    B+3,X'F0'            And the old sign position
         - - -
B        DS    0PL5                 5-byte result = 000123400+
A        DC    PL4'1234'            Initial value of number to shift
         DS    P                    Reserve one extra byte
```

Figure 318. Shifting a decimal operand left an even number of digits

As these two examples show, extra effort is needed to zero out the positions vacated by the “shift”. It may be simpler in some cases to use an SS-type instruction to set the necessary digit positions to zero, as in Figure 319.

```
         MVC   A(3),A+1             Move three bytes left by 1 byte
         NC    A+2(2),Mask          Set 2 vacated digits to zero
         - - -
A        DC    PL4'1234'            Initial operand
Mask     DC    X'F00F'              Mask for zeroing 2 vacated digits
```

Figure 319. Shifting a decimal operand left an even number of digits

This NC instruction does the same zeroing as the two NI instructions in Figure 318.

29.9.5. Shifting Right an Even Number of Digits

To shift *right* an even number of places, two techniques are used, depending on whether or not the result must have the same length as the original operand. If the result may have a different length, it is simplest to move the sign digit to the left, and leave the leftover data in place. For example, suppose 12345+ is to be shifted right two digit positions, so 123+ is the desired result.

```
         MVN   A+1(1),A+2           Move sign 2 digits to left
         - - -
B        DS    0PL2                 For referring to shifted result
A        DC    PL3'12345'           Final contents = 123+5+
```

Figure 320. Shifting a decimal operand right an even number of digits

We must now refer to the result at **B** with its new name and length attribute, since the original three-byte operand now contains an invalid digit. (See Exercise 29.9.4.)

When the length of the original operand must be preserved (so that the desired number of high-order zeros actually appears in the leftmost bytes), it is difficult to use the move instructions, because they process data from left to right, rather than from right to left.

Surprisingly, it is easiest to shift right an *even* number of digit positions by doing two *odd* shifts! To illustrate, we can shift 12345+ to the right by two digit positions using two MVO instructions, as in Figure 321.

```
         MVO   A(3),A(2)          Shift once, result = 01234+
         MVO   A(3),A(2)          Shift again, result = 00123+
         - - -
A        DC    P'12345'           Original 3-byte operand = 12345+
```

Figure 321. Shifting a decimal operand right an even number of digits

As these examples show, SRP may be much simpler; it also provides protection against invalid data and unexpected decimal overflow.

Exercises

29.9.1.(1)+ At the end of Section 29.9.3, the text states that we can't use the MP instruction to multiply 0001234+ by 100+. Explain why not.

29.9.2.(3) Generalize the instructions in Figure 314 on page 519 to handle an odd left shift of N digit positions of the L-byte operand at **A**. State any restrictions that must be imposed on the values of L and N.

29.9.3.(3) Generalize the instructions in Figure 316 on page 519 to an odd left shift of N digit positions of the L-byte operand at **A**, leaving the shifted result at **B**. State any restrictions imposed on the values of L and N.

29.9.4.(3) Generalize the technique illustrated in Figure 320 on page 520 to an L-byte operand shifted right an even number N of digit positions. State any restrictions on N and L.

29.9.5.(2) In the discussion preceding Figure 313 on page 518 about using MVO to shift an L-byte operand to the right by an odd number of digit positions N, we said that we required $N < 2L-1$. Why can't this requirement be written $N \leq 2L-1$?

29.9.6.(2) Write a sequence of instructions using the technique illustrated in Figure 315 on page 519 to perform a single left shift of the L-byte packed decimal operand at **A**. If the shift causes a decimal "overflow", branch to **Over** after the shift is completed.

29.9.7.(3) The decimal operand at **A** has length L bytes, and is to be shifted left N digit positions, where N is an even number. Generalize the instruction sequence shown in Figure 319 on page 520 to do the shift. State any restrictions that must be placed on N and L.

29.9.8.(3) Do the same as in Exercise 29.9.7, but branch to **Over** if significant high-order digits are lost.

29.9.9.(3) Using the same conditions as in Exercise 29.9.7, generalize the instruction sequence of Figure 318 on page 520, and state any necessary restrictions on N and L.

29.9.10.(3) Suppose we wish to multiply the decimal number at **A** by an *even* power of ten, where the power of ten is defined in an EQU statement such as

```
N        Equ   6                  Multiply by 10**6
```

Write a sequence of instructions using the techniques illustrated in Figures 317 through 319 to perform the desired multiplication. What restrictions must be placed on the value of N, and how will it depend on the length of the number at **A**?

29.9.11.(5) Write instruction sequences not using SRP that will shift a decimal number to the right N digit positions, and *round* the result properly. (Rounding should be done by adding 5 to the most significant digit shifted out of the operand.)

29.9.12.(3) Generalize the technique in Figure 321 to shift an L-byte operand to the right by an even number N of digit positions. State any restrictions on L and N.

29.10. Scaled Packed Decimal Computations: General Rules

All previous examples of packed decimal arithmetic have assumed integer operand values. But decimal arithmetic must often deal with numbers containing fractional parts, such as currency values, percentages, and the like. Packed decimal arithmetic with such values requires some extra considerations.

For example, adding two currency values like $123.45 and $234.56 is simple; we can treat them as integers, and remember to insert the decimal point in the right place when the result is formatted for printing or display.

```
  $123.45
+ $234.56
  $358.01
```

Similarly, multiplying a currency value by an integer can be done in much the same way:

```
  $123.45
x      12
 $1481.40
```

But if our data is more complex, we need to know more.

29.10.1. Precision and Scale

There are two senses of the word “precision”.

- In Assembler Language terms, precision refers to the number of digits a field or register can hold. A 4-byte object can hold 32-bit integers, or 7 packed decimal digits, or 4 zoned decimal digits. This is sometimes called “field” precision. We don't need to know the values stored in those objects.
- In computational terms, precision refers to the number of *significant* digits in a number. A value like 3.14159265358979 has 15 significant digits, so its precision is 15 digits. The number 3.12345678901234 is also precise to 15 digits.[189]

Because we may not know the precision of a *value* stored in a field at any given time, we must work with the known maximum precision of the value: the precision of the *field*.

Overflows can also cause inaccuracy in precise values: the sum of two 32-bit integers that causes a fixed-point overflow is still precise to 32 bits, but a very inaccurate representation of the true sum.

We must often do arithmetic with numbers having different numbers of digits before and after the decimal point. For example, suppose we use an AP instruction to directly add two 3-digit packed decimal numbers:

```
A        DC    P'123.'              X'123C'
B        DC    P'.456'              X'456C'
```

Because packed decimal arithmetic is *integer* arithmetic, irrespective of any decimal points you may have put in the nominal values, we find that the sum P'579' has no correct digits. For a correct sum, we must first define a field to hold 6 or more digits:

```
Sum      DS    PL4                  Space for 7 digits
```

[189] The first value is an approximation to pi that is also *accurate* to 15 decimal digits. However, the second value, as an approximation to pi, is accurate to only two digits. There's a big difference between precision and accuracy!

To account for the differences in decimal point positions, we must (1) align the operands and (2) remember the position of the decimal point in the sum:

```
         ZAP   Sum,A            c(Sum) = X'0000123C' = 123.
         SRP   Sum,64+3,0       c(Sum) = X'0123000C' = 123.000
         AP    Sum,B            c(Sum) = X'0123456C' = 123.456
         - - -
A        DC    P'123.'
B        DC    P'.456'
Sum      DS    PL4              Space for 7 digits
```

Figure 322. Ensuring decimal point alignment for packed decimal addition

It is up to you to write instructions that keep track of the position of the decimal in all mixed integer-fraction calculations.[190]

We will use "I.F" as our notation for a number containing both integer and fraction values, where I is the number of integer digits and F is the number of fraction digits.

29.10.2. General Rules: Addition and Subtraction

Consider adding these two numbers:

```
   123.45
 + 994.7264
  1118.1764
```

Because the sum may cause a carry from the high-order digit of the larger operand, the I value of the result is one larger than the I value of the larger operand. (For subtraction, the result could have fewer digits.)

If we call 123.45 the first operand, its I and F values are $I_1=3$ and $F_1=2$; for the second operand, $I_2=3$ and $F_2=4$. We see that the result's I and F values are F=4 and I=4.

We can infer a general rule for addition and subtraction:

$$I = Max(I_1, I_2) + 1$$
$$F = Max(F_1, F_2)$$

That is, a sum or difference may have one more integer digit than the larger number of integer digits of the operands, and it may have the larger number of fraction digits of the two operands.

29.10.3. General Rules: Multiplication

We'll use another example to illustrate a general rule. Suppose we multiply these two numbers:

```
    432.45
 x    6.0743
   2626.831035
```

and we see that the product's number of integer and fraction digits is given by

$$I = I_1 + I_2$$
$$F = F_1 + F_2$$

These values are of course the largest possible; if we multiply 1.25×2.2=2.750 we see that I for the product can be less than the maximum; and if we omit trailing zeros, F can also be less than the maximum.

[190] The Integer (I') and Scale (S') attributes of packed decimal operands can help; see the example in Section 29.11.4.

29.10.4. General Rules: Division (*)

Division rules are more complicated than those for addition, subtraction, and multiplication. To start, consider the simple division 4/3:

```
4./3. = 1.33333333333333333333333333333333333333333333333...
```

We see that the number of integer digits is limited, but the unending fraction is a problem. So we need a rule to give useful values of I and F. First, we'll consider I:

```
4./.3 = 13.3333333333333333333333333333333333333333333333...
4./.03 = 133.333333333333333333333333333333333333333333333...
4./.003 = 1333.33333333333333333333333333333333333333333333...
4./.0003 = 13333.3333333333333333333333333333333333333333333...
```

so we see that the value of I is

$$I = I_1 + F_2$$

Depending on how many fraction digits we want to keep in the result, the length of the packed decimal quotient field might need to be much larger than I to accommodate the quotient's fraction digits.

As another example, suppose we divide 457.9 by 1.23. Since multiplying both terms by the same number doesn't change the quotient, we can multiply both by 100 to eliminate the fraction parts, so that the divisor (1.23) is now 123 and the dividend is now 45790:

```
45790/123 = 372.27642276422764227642276422764227642276...
```

where the underscore indicates that the group of those digits repeats endlessly.

The unending quotient is again a problem: how many fraction digits should we keep? Usually we will round the result to a "reasonable" number of digits, being careful to keep correct numbers of integer and fraction digits. But, what *is* the correct number of digits?

You can consider two ways to handle this question:

1. If the fraction digits aren't important, you can use the shortest valid lengths for divisor and dividend. For example, in evaluating 4/3 we could write:

```
         ZAP   Num,=P'4'          Set up dividend
         DP    Num,Den            Divide by divisor
         - - -
Num      DS    PL2                Minimum possible length!
Den      DC    P'3'
```

   and the result at **Num** will be X'1C1C', with quotient 1 and remainder 1.

2. If, however, you want to retain (say, 5) digits of the fraction, you will first need to allocate a longer area for **Num** and then multiply the dividend by 10^5:

```
         ZAP   Num,=P'4'          Set up dividend
         SRP   Num,64+5,0         Scale up by 5 digits (* 10000)
         DP    Num,Den            Divide by divisor
         - - -
Num      DS    PL5                Space for quotient and remainder
Den      DC    P'3'
```

   and the result at **Num** will be X'0133333C1C', with quotient 133333 and remainder 1. When you format the result for printing or display, you will have to "unscale" the result by placing the decimal point correctly. For example, you can use the ED (Edit) instruction (to be described in Section 30):

```
         MVC   W(L'EdM),EdM       Copy edit pattern
         ED    W,Num              Edit the result
         - - -
EdM      DC    X'4020214B2020202020' bds.ddddd    7 quotient digits
W        DS    CL9
```

   and the result at **W** is C'••1.33333' (where • represents a space).

As another example, suppose we want to print the quotient of 479.6/1.23, rounded to 2 decimal places. Now,

```
479.6/1.23 = 389.918699186991869918699186991869918699...
```

so our final rounded result should be 389.92. First, we must multiply by 100 to change both operands to integers, so the division becomes 47960/123, which has the same result. The quotient and divisor both have 3 digits (which fit in 2-byte packed decimal numbers), so we might try this:

```
         ZAP   Num,=P'47960'      Set up dividend
         DP    Num,Den            Divide by divisor
         - - -
Num      DS    PL4                Space for quotient and remainder
Den      DC    P'123'
```

which gives a result at **Num** of X'389C113C', a quotient of 389 and remainder 113. But this doesn't have enough digits for both the fraction and the remainder parts (we want 2 rounded fraction digits), so we need at least one additional digit for rounding. So, we first multiply the numerator by 1000, remembering that we must increase the length of the **Num** field by 2 additional bytes:

```
         ZAP   Num,=P'47960'      Set up dividend
         SRP   Num,64+3,0         Scale up by 3 digits
         DP    Num,Den            Divide by divisor
         - - -
Num      DS    PL6                Space for quotient and remainder
Den      DC    P'123'
```

which gives a result at **Num** of X'0389918C086C'. The quotient, X'389918C', is ready for rounding using an SRP instruction:

```
         ZAP   Num,=P'47960'      Set up dividend
         SRP   Num,64+3,0         Scale up by 3 digits
         DP    Num,Den            Divide by divisor
         SRP   Num(4),64-1,5      Shift right one digit and round
         - - -
Num      DS    PL6                Space for quotient and remainder
Den      DC    P'123'
```

and the rounded result at **Num** is X'0038992C086C' (where the rounded digit 2 is underscored) , ready for printing to 5 significant digits.

The answer to our question "what *is* the correct number of digits?" depends on how many fraction digits are required for a particular application. If we say that the total number of integer and fraction digits needed is N=I+F, and we already know the value of

$$I = I_1 + F_2,$$

then for division the number of fraction digits F is simply

$$F = N - I$$

29.10.5. COBOL and PL/I Notations (*)

High-level languages like COBOL and PL/I keep track of these details for you. Each numeric quantity is explicitly or implicitly declared with a number of integer and fraction digits, and the compiler and its run-time library use that information to produce valid results whenever possible.

They use different terminology than the "I.F" described above:

- COBOL calls the number of integer digits *integers* and the number of fraction digits *decimals*. Thus,

  ```
  01  Number Pic S9(4)V9(2) Packed-decimal Value +1234.56
  ```

  declares **Number** to have 4 integers and 2 decimals.

- PL/I assigns a *precision* p and a *scale* q to fixed-point values: p is the total number of digits the field can hold, and q is the number of fraction digits. Thus,

  ```
  Declare Number Decimal (6,2) Init (1234.56);
  ```

  assigns p=6 and q=2 to **Number**. Thus for PL/I, I.F = (p−q).q.

Exercises

29.10.1.(2) Express the I and F values of the three symbols in Figure 322 on page 523 in PL/I notation.

29.11. Example of a Packed Decimal "Business" Computation

To illustrate a "real" packed decimal application, suppose a wholesaler and a retailer complete an order:

1. The retailer orders from the wholesaler 60 high-tech widgets at $74.65 each.
   (60×$74.65 = $4479.00)
2. For this large order, the wholesaler discounts the price by 4.7%.
   ($4479.00×0.047 = $210.513, so the discounted price is $4479.00−$210.51 = $4268.49)
3. The wholesaler adds 9.75% local sales tax, and a $4.00 per-item shipping charge.
   (4268.49×1.0975 = 4684.67; the shipping charge is 60×$4.00 = $240.00, so the total is $4684.67+$240.00 = $4924.67.)
4. The retailer's pre-payment of $1000.00 is deducted. The result is the wholesaler's bill to the retailer.
   ($4924.67−$1000.00 = $3924.67)

Then, the retailer calculates his necessary markup and the sale price:

5. The retailer calculates his base cost for each item.
   ($4924.67/60 = $82.07785 or $82.08)
6. He then applies his retail markup (about 37%), adjusted to a sale cost just below one dollar.
   (The markup is $82.08×1.37 = $112.4496 or $112.45, and the adjustment is $0.54+$112.45 = $112.99)
7. Each item a customer buys must include 9.25% sales tax and a $7.50 recycling fee.
   (The price including sales tax is $112.99×1.0925 = $123.44; adding the recycling fee gives the final customer cost: $123.44+$7.50 = $130.94)
8. After making the "cents" portion of the price 99 cents, the result is the final cost per item to the customer.
9. The retailer's gross profit per item is (sale cost)−(base cost).
   ($112.99−$82.08 = $30.91. The percent gross profit is $30.91/$82.08 = 0.376 or 38%)

We will now see how each step is calculated in packed decimal arithmetic.

29.11.1. The Wholesaler's Calculation

First, he creates data and work areas, choosing field lengths to hold values and results. The parenthesized values are (**I.F**): the number of integer digits and the number of fraction digits. Thus, the width of the field is I+F *digits*. (Remember that the number of *bytes* is (I+F)/2+1.).

```
UnitCost DC    PL3'74.65'        Base cost per item                 (3.2)
WItems   DC    PL2'60'           Number of wholesale items ordered  (3.0)
Discount DC    PL2'.047'         Discount percentage for large order (0.3)
WhlseTax DC    PL3'.0975'        Sales tax at wholesaler location   (1.4)
ShipChrg DC    PL2'4.00'         Shipping charge per item           (1.2)
Prepaid  DC    PL5'1000.00'      Retailer's prepayment              (7.2)
WhlseNet DS    PL5'00000.00'     Net cost to retailer               (7.2)
WWorkVal DS    PL8               Work area for wholesaler calculations
ShipWork DS    PL4               Work area for shipping charge
WDisc    DS    PL8               Work area for bulk-order discount
WSTax    DS    PL8               Work area for wholesale sales tax
```

Figure 323. A business calculation in packed decimal, part 1

Next, the wholesaler uses this data and the work areas to calculate the retailer's bill.

```
     ZAP   WWorkVal,UnitCost  Set up unit cost              X'000000000007465C' (13.2)
*                                 (nnnnnnnnnnnn.nn)
     MP    WWorkval,WItems    Multiply by no.items ordered  X'000000000447900C' (13.2)
```

Because the number of items is an integer, there was no change in the number of fraction digits in the product.

```
     ZAP   WDisc,WWorkVal     Copy base cost for bulk discount
     MP    WDisc,Discount     Discounted cost               X'000000021051300C' (13.5)
     SRP   WDisc,64-3,5       Rounded bulk discount         X'000000000021051C' (13.2)
     SP    WWorkVal,WDisc     Calculate discounted cost     X'000000000426849C' (13.2)
```

The discount has 3 fraction digits, so the number of fraction digits in the product increases to 5, so we shifted right 3 places and round to get the correct discount.

```
     ZAP   WSTax,WWorkVal     Copy for calculating sales tax
     MP    WSTax,WhlseTax     Wholesaler's sales tax        X'000000416177775C' (11.6)
*                                 (nnnnnnnnn.nnnnnn)
     SRP   WSTax,64-4,5       Calculate rounded sales tax   X'000000000041618C' (13.2)
     AP    WWorkval,WStax     Add wholesaler's sales tax    X'000000000468467C' (13.2)
```

The sales tax has 4 fraction digits, so the number of fraction digits in the product increases to 6, so we shifted right 4 places and round to get the correct tax amount.

```
     ZAP   ShipWork,ShipChrg  Set shipping charge
     MP    ShipWork,WItems    Times number of items         X'0024000C'         (5.2)
     AP    WWorkVal,ShipWork  Add shipping charge           X'000000000492467C' (13.2)
     SP    WWorkVal,PrePaid   Subtract retailer's prepayment X'000000000392467C' (13.2)
     ZAP   WhlseNet,WWorkVal  Now have net cost to retailer X'000392467C'       (7.2)
```

Figure 324. A business calculation in packed decimal, part 2

Thus, the wholesaler's bill to the retailer is `X'000392467C'`, or $3924.67. Note that the instructions must keep track of the decimal point at each step.

29.11.2. The Retailer's Calculation

Now that the retailer has received the widgets and the bill from the wholesaler he must determine what to charge as a retail price, and what his margin of profit will be for each widget he sells.

First, he creates data and work areas, choosing field lengths to hold values and results:

```
WhlseChg DC    PL4'4924.67'       Charge for the full order          (5.2)
NItems   DC    PL2'60'            Number of items ordered            (3.0)
BaseCost DS    PL4'0'             Base cost per item work area
RMarkup  DC    PL3'1.37'          Basic retail markup margin         (5.2)
Retail   DS    PL4                Retail advertised price per item
GrossPrf DS    PL3'0'             Gross profit per item sold
CustCost DS    PL4                Final charge per item to customer
RWorkVal DS    PL8                Work area for retail calculations
RetailTx DC    PL3'.0925'         Sales tax at retailer location     (1.4)
Recycle  DC    PL3'7.50'          Recycling charge per item          (3,2)
ItemCost DS    PL4                Total customer cost per item
```

Figure 325. A business calculation in packed decimal, part 3

As in the wholesaler's calculation, the (precision,scale) of each item are shown.

The retailer can now calculate the retail cost per widget and his expected rate of profit per item.

```
      ZAP   RWorkVal,WhlseChg  Prepare to calculate base cost  X'000000000492467C' (13.2)
      DP    RWorkVal,NItems    Calculate base cost per item    X'000000008207C047C' (9.2)
```

The retailer wants to round the base cost, but SRP won't work: only if twice the remainder is greater than or equal to the number of items (the divisor) should the quotient be rounded.

```
      AP    RWorkVal+6(2),RWorkVal+6(2) Check remainder        X'000000008207C094C' (1.2)
      CP    RWorkVal+6(2),NItems    2*Remainder < NItems?
      JL    NoRnd              Branch if yes, no roundup
      AP    RWorkVal(6),=P'1'  Round up base cost              X'00000008208C'      (9.2)
NoRnd ZAP   RWorkVal,RWorkVal(L'RWorkVal-2) Drop remainder
      ZAP   BaseCost,RWorkVal  Save the base cost per item     X'0008208C'          (5.2)
      MP    RWorkVal,RMarkup   Find approximate retail price   X'000000001124496C' (11,4)
      SRP   RWorkVal,64-2,5    Round off the two extra digits  X'000000000011245C' (13.2)
      NC    RWorkVal+L'RWorkVal-2(2),=X'F00F' Delete 2 digits  X'000000000011200C' (13.2)
      AP    RWorkVal,=P'99'    Force price to end in .99       X'000000000011299C' (13.2)
```

He wanted the sales price to seem more “sale-like”, so he added .99 to force the sales price upward to end in .99.

```
      ZAP   Retail,RWorkVal    Save advertised retail price    X'0011299C'          (5.2)
      MP    RWorkVal,RetailTx  Calculate retailer's sales tax  X'000000010451575C' (9.6)
      SRP   RWorkVal,64-4,5    Round off to dollars/cents      X'000000000001045C' (13.2)
```

Just as the wholesaler did, the retailer compensates for the four fraction digits in the sales tax by rounding the tax to have two decimal digits.

```
      AP    RWorkVal,Retail    Add the retail cost             X'000000000012344C' (13.2)
      AP    RWorkVal,Recycle   Add widget recycle charge       X'000000000013094C' (13.2)
      ZAP   CustCost,RWorkVal  Save customer cost/item         X'0013094C'          (5.2)
      ZAP   RWorkVal,Retail    Calculate gross profit per item
      SP    RWorkVal,BaseCost  Subtract base from retail       X'000000000003091C' (13.2)
      SRP   RWorkVal,3,0       Position for % calculation      X'000000003091000C' (10.5)
      DP    RWorkVal,BaseCost  Percentage gross profit         X'0000376C0004792C' (7.1)
```

Figure 326. A business calculation in packed decimal, part 4

The third and fourth instructions help calculate a rounded quotient; like binary division, packed decimal quotients are unrounded. If 2×(remainder) ≥ divisor, we add 1 to round the quotient.

Finally: the advertised retail cost per widget is $112.99, and with sales tax and recycling charge, total cost per widget to the customer is X'0013094C' = $130.94.

The retailer's gross profit per item is X'0003091C' = $30.91, and his profit margin is X'0000376C' = 37.6%.

29.11.3. Comments

As these examples illustrate, doing “realistic” calculations with packed decimal arithmetic can be complicated; you must plan carefully to ensure that field lengths are correct and that edit patterns (described in Section 30) produce correctly formatted results. This “business” calculation can be done *much* more easily in Assembler Language using Decimal Floating-Point, as we'll see in Section 35.13; but (for packed decimal arithmetic) many programmers prefer letting a high level language like PL/I or COBOL worry about the details.

Preview of Coming Attractions

Having completed this section, you can understand the difficulties of typical packed decimal arithmetic calculations. Chapter IX will describe Decimal Floating-Point arithmetic and data, which greatly simplifies the scaling problems inherent in packed decimal.

29.11.4. Using Integer and Scale Attributes (*)

The assembler can help with packed decimal data containing integer and/or fraction digits, using the Integer Attribute (I') and the Scale Attribute (S') of a symbol. Consider the symbols in Figure 322 on page 523:

```
         ZAP   Sum,A              c(Sum) = X'0000123C' = 123.
         SRP   Sum,64+3,0         c(Sum) = X'0123000C' = 123.000
         AP    Sum,B              c(Sum) = X'0123456C' = 123.456
         - - -
A        DC    P'123.'
B        DC    P'.456'
Sum      DS    PL4                Space for 7 digits
```

The values of the symbol attributes are shown in Figure 327, where we see that I'A=3, S'A=0, I'B=0, S'B=3, I'Sum=7 and S'Sum=0.

```
000000 123C        1 A        DC    P'123.'
000002 456C        2 B        DC    P'.456'
000004             3 Sum      DS    PL4

000008 0300        5          DC    AL1(I'A,S'A)
00000A 0003        6          DC    AL1(I'B,S'B)
00000C 0700        7          DC    AL1(I'Sum,S'Sum)
```

Figure 327. Integer and Scale Attributes

We could use these values in the SRP instruction to determine the amount of the shift:

```
         ZAP   Sum,A                c(Sum) = X'0000123C'     (7.0)
         SRP   Sum,64+(S'B-S'A),0   c(Sum) = X'0123000C'     (4.3)
         AP    Sum,B                c(Sum) = X'0123456C'     (4.3)
```

Figure 328. Using Scale Attributes in a SRP instruction

In practice, Integer and Scale attributes are used rather rarely, and then mainly in macro instructions.

Exercises

29.11.1.(2) Show the Integer and Scale Attributes of the following packed decimal data items:

1. `P'1024.2048'`
2. `P'-0.98765'`
3. `P'+2235058.4'`
4. `P'72.3456'`

29.12. Summary

Some properties of the instructions discussed in this section are summarized in Table 216.

Mnemonic	Generate preferred signs?	Possible interruptions
AP	Y	Decimal overflow, decimal data
CP	N	Decimal data
DP	Y	Zero divide, decimal data, specification
MP	Y	Decimal data, specification
MVO	N	—
SP	Y	Decimal overflow, decimal data
SRP	Y	Decimal overflow, decimal data
TP	N	—
ZAP	Y	Decimal overflow, decimal data

Table 216. Summary of decimal instruction behavior

Terms and Definitions

biased rounding
: A common type of rounding that introduces a small inaccuracy in the rounded results; typically caused by rounding decimal values by adding 5 to the last discarded digit.

offset
: The MVO instruction shifts or *offsets* the second operand to the left by one hex digit before appending it to the left of the sign digit of the first operand.

operand order dependence
: The results of many packed decimal arithmetic instructions depend on the order of the operands. For example (007+)+(7+) yields 014+, but (7+)+(007+) causes a decimal overflow.

precision
: Precision can be defined in two ways. (1) The number of digits that can be held in a register or memory field. (2) The number of significant digits in a numeric value.

scaled arithmetic
: Methods for doing arithmetic with non-integer values, using fixed-point arithmetic instructions such as binary integer and packed decimal.

unbiased rounding
: A technique for avoiding the inaccuracies introduced by biased rounding, typically by rounding results exactly half way between two representable results to the value with an even low-order digit.

Programming Problems

Problem 29.1.(4) Rewrite your solution to Problem 25.2 on page 398 to do perfect shuffles using packed decimal numbers throughout.

Problem 29.2.(1)+ In Section 28.2 on page 486, the text states that (5−)+(5−) will generate 0−. Write a short program to show that this is true.

Problem 29.3.(1)+ In Section 28.4 on page 489, the text states that (2+)×(0−) will generate 0−. Write a short program to show that this is true.

30. Converting and Formatting Packed Decimal Data

```
 3333333333      00000000
333333333333    0000000000
33         33  00        00
           33  00        00
           33  00        00
        3333   00        00
        3333   00        00
           33  00        00
           33  00        00
33         33  00        00
333333333333    0000000000
 3333333333      00000000
```

The instructions in Table 217 perform useful operations on packed decimal numbers. Along with PACK and UNPK, they are used to convert data among binary, packed decimal, and character formats.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
4F	CVB	RX	Convert to Binary (32)	4E	CVD	RX	Convert to Decimal (32)
EB06	CVBY	RXY	Convert to Binary (32)	E326	CVDY	RXY	Convert to Decimal (32)
E30E	CVBG	RXY	Convert to Binary (64)	E32E	CVDG	RXY	Convert to Decimal (64)
DE	ED	SS	Edit	DF	EDMK	SS	Edit and Mark

Table 217. Instructions used for converting and formatting packed decimal

The only difference between CVB and CVBY, and between CVD and CVDY, is that the instructions ending with "Y" have signed 20-bit displacements, while the other two have unsigned 12-bit displacements.

30.1. CVD, CVDY, and CVDG Instructions

The CVD and CVDY instructions convert a two's complement binary number in the rightmost 32 bits of general register R_1 to packed decimal format and store it at the 8-byte[191] second operand in memory. To illustrate, suppose c(GR7)=X'00000087', or +135 in decimal. If we execute either of the CVD or CVDY instructions

```
         CVD   7,WorkArea          Convert to packed decimal at WorkArea
         CVDY  7,WorkArea          Convert to packed decimal at WorkArea
         - - -
WorkArea DS    D                   Result = X'00000000 0000135C'
```

then the packed decimal result at **WorkArea** will have the value shown.

[191] On System/360 CPUs, the second operand was *required* to be aligned on a doubleword boundary; if it wasn't, a specification exception caused a program interruption. Like many other operand alignment requirements, it was removed in System/370, but doubleword alignment is still a good practice.

Because the second operand field can hold 15 decimal digits, there will always be at least five high-order zeros in the result. The largest magnitude of the first operand is −2147483648, which has ten significant digits. When converted to packed decimal, the result is X'000002147483648D'.

Similarly, CVDG converts the 64-bit binary integer first operand in GG R_1 to a 16-byte packed decimal second operand. Thus, if GG8 contains X'4000000000000000' (2^{62}),

```
         CVDG  8,WrkArea2          Convert to 16-byte packed decimal
         - - -
WrkArea2 DS    2D                  Result=X'00000000 00004611 68601842 7387904C'
```

Figure 329. Converting a 64-bit binary integer to packed decimal

then the packed decimal result at **WrkArea2** has the value shown.

These three instructions may be thought of as “store” instructions, because the data “moves” from the first operand (a register) to the second (in memory).

Suppose we use CVD to help produce printable decimal values, as in a program that prints a multi-page listing with a page number on each page. Suppose the halfword integer at **PgNum** contains the current page number, and we want to put that number (preceded by the characters “Page”) into an area named **PageNo**.

```
NDigits  Equ   5                   Max number of digits in page number
         LH    0,PgNum             Get binary value of page number
         CVD   0,WorkArea          Convert to decimal
         UNPK  ZonePgN,WorkArea    Unpack to zoned format
         OI    ZonePgN+L'ZonePgN-1,C'0'  Set zone X'F' for last digit
         - - -
WorkArea DS    0D,XL8              Conversion work area
PgNum    DC    H'345'              Simulated page number
PageNo   DS    0CL(5+NDigits)      Length for 'Page nnnnn'
         DC    C'Page '
ZonePgN  DS    ZL(NDigits)         Character form of page number
```

Figure 330. Using CVD to format page numbers

UNPK transfers the original sign of the packed operand to the zoned operand, which would be X'C' in this example. Thus, the OI instruction sets the zone digit in the low-order character of the field, for its correct EBCDIC representation.

This instruction sequence has a minor defect, however: if the number to be printed has fewer significant digits than the size of the field, the result will contain leading zeros. A simple loop can be written to change leading zeros to blanks, but the ED and EDMK instructions provide a more elegant and powerful solution to this problem. (See Exercises 30.1.4 and 30.1.5.)

Exercises

30.1.1.(1)+ In Figure 329, the result at **WrkArea2** has many leading zero digits. What is the minimum number of zero hex digits in the result from a CVDG instruction?

30.1.2.(1) Why can the 8-byte second operand of a CVD instruction hold 15 digits?

30.1.3.(1) Suppose GR1 contains the maximum negative number X'80000000', and the instruction

```
         CVD   1,X
```

is executed. Show the contents of 8-byte area named **X**.

30.1.4.(2)+ Suppose the OI instruction had been omitted in Figure 330. Explain what would happen when the result at **PageNo** was printed, given that the page number at **PgNum** was first 29, then 30, and then 31.

30.1.5.(2)+ Add a short loop to the instruction sequence in Figure 330 to blank the leading zeros in the page number.

30.1.6.(2)+ For each of these values at **Arg**,

```
 1. Arg     DC    F'-1'
 2. Arg     DC    F'2147483321'
 3. Arg     DC    X'77777783'
 4. Arg     DC    C'   *'
```

show the packed decimal result at **DecVal** after executing these instructions:

```
         L     1,Arg
         CVD   1,DecVal
         - - -
DecVal   DS    0D,XL8
```

30.1.7.(3)+ For each of these values at **Arg2**,

```
 1. Arg2    DC    FD'-1'
 2. Arg2    DC    FD'9223372036854775807'
 3. Arg2    DC    XL8'B7777783A60D93CA2'
 4. Arg2    DC    CL8'Ghastly!'
```

show the packed decimal result at **DecVal2** after executing these instructions:

```
         LG    1,Arg2
         CVDG  1,DecVal2
         - - -
DecVal2  DS    0D,XL16
```

30.1.8.(4) Write a sequence of instructions to simulate the operation of CVD.

30.2. CVB, CVBY, and CVBG Instructions

The CVB and CVBY instructions perform the inverse operation to CVD and CVDY; similarly, CVBG does the inverse of CVDG. That is, the packed decimal number at the second operand address is converted to two's complement binary form and the result is placed in general register R_1. Thus, they can be thought of as “load” instructions that “move” data from the second operand (in memory) to the first (in a register).

To illustrate, suppose we execute this CVB instruction:

```
         CVB   6,PackNum            Result in GR6 = X'FFFFFF79'
         - - -
PackNum  DC    PL8'-135'            X'000000000000135D'
```

The result in GR6 is X'FFFFFF79', with value −135.

Similarly, if we execute this CVBG instruction

```
         CVBG  9,PackNum2           Convert to 64 bits in GG9
         - - -
PackNum2 DC    PL16'-123456789012345'
```

The result in GG9 is X'FFFF8FB779F22087'.

All three instructions are subject to two possible program interruption conditions.

- If the second operand field does not contain valid decimal data, a decimal data exception occurs and the Interruption Code is set to 7.
- If the value of the decimal operand is too large to fit in the first operand register, a fixed-point divide exception occurs and the IC is set to 9.

This situation is handled differently for CVB/CVBY and CVBG.

- For CVB and CVBY, the low-order 32 bits of the two's complement binary result are placed into register R_1.
- For CVBG, the operation is suppressed, and register R_1 is unchanged.

Suppose an 80-byte record contains numeric data, and we must convert a number on the record from character form to binary and place the result in GR5. Assume the characters are right-justified in an 8-byte field of the record at offset **DOff**.

```
DLen     Equ   8                  Length of data field
DOff     Equ   72                 Offset of start of field
         PACK  DWork,Record+DOff(DLen) Pack data to decimal
         CVB   5,DWork            Convert to binary value in GR5
         - - -
DWork    DS    0D,XL8             Conversion area
Record   DC    CL75' ',C'45678'   80-byte input data record
```

Figure 331. Converting decimal characters to binary

This technique is adequate for unsigned numbers, or for zoned decimal data in which the sign is "over-punched" in the last digit column of the input field. Greater care is needed when the digits may be preceded by a sign. (See Exercise 30.2.6.)

Exercises

30.2.1.(1)+ What will happen if a CVB instruction specifies a second operand containing X'000002147483648C'?

30.2.2.(2)+ Suppose we execute the following two instructions. What will happen, and what results will be found at **DWork** and in GR0?

```
         PACK  DWork,Data
         CVB   0,DWork
         - - -
DWork    DS    D
Data     DC    C'123'
```

30.2.3.(2)+ What will happen in the instruction sequence of Exercise 30.2.2 if **Data** is defined by

```
Data     DC    CL20'123' ?
```

30.2.4.(2)+ What will happen in the instruction sequence of Exercise 30.2.2 if **Data** contains four blanks, such as

```
Data     DC    CL4' ' ?
```

30.2.5.(4) If the binary result of a CVB instruction is too large, an interruption occurs after the low-order 32 bits of the result are put into the first operand register. This means that a certain amount of internal arithmetic apparently had to be performed to a precision of *more* than 32 bits; what is the minimum number of internal bits required?

30.2.6.(3)+ Suppose the datum in the 8-column field in Figure 331 might be preceded by an optional sign character, C'+' or C'−'. Modify the instruction sequence to place the correctly signed binary result into GR5. (Assume that the number is well-formed: it contains no extra signs, no embedded or trailing blanks, no invalid characters, etc.)

30.2.7.(1) Why do you think the designers of the CVB instruction chose to place the low-order 32 bits of the result into general register R_1 even when the fixed-point divide exception indicates that the result is not correct?

30.2.8.(2) Show the resulting value in GR0 for each of these data values at **PackData** after executing this instruction:

```
         CVB   0,PackData
```

1. `PackData DC    XL8'FFFFFFFC'`
2. `PackData DC    XL8'743915506D'`
3. `PackData DC    XL8'AFFFFFFFFFFFFFFD'`
4. `PackData DC    XL8'999C'`

30.2.9.(3) Write a sequence of instructions to simulate the operation of CVB.

30.2.10.(3) Repeat Exercise 30.2.6, but make no assumptions about correctly formed input data. If an invalid character is found, branch to **BadChar** with its address in GR1.

30.3. Editing Overview

The ED and EDMK instructions are complex. They are related to UNPK: they convert the packed decimal second operand to character form in the first operand. However, that is about *all* that is similar between UNPK and the editing instructions! Editing can also suppress leading zeros, insert special characters, and do other things that produce readable results.

The Assembler Language syntax for both instructions is

mnemonic $D_1(N,B_1),D_2(B_2)$

or

mnemonic Pattern(N),PackData

Both editing instructions are single-length SS-type instructions, as shown in Table 218.

opcode	L	B_1	D_1	B_2	D_2

Table 218. Format of the ED and EDMK instructions

The length field of these instructions refers only to the length of the *first*, or *pattern*, operand (unlike other single-length SS-type instructions in which the lengths of the two operands are considered to be the same). The number of bytes taken from the second operand depends on the contents of both operands.

Editing converts the signed or unsigned[192] packed decimal second operand to character (zoned) form under the control of a *pattern* provided by the first operand, as sketched in Figure 332. The bytes of the pattern provide a *picture* showing what the edited result should look like: each byte in the pattern represents one character of the result.

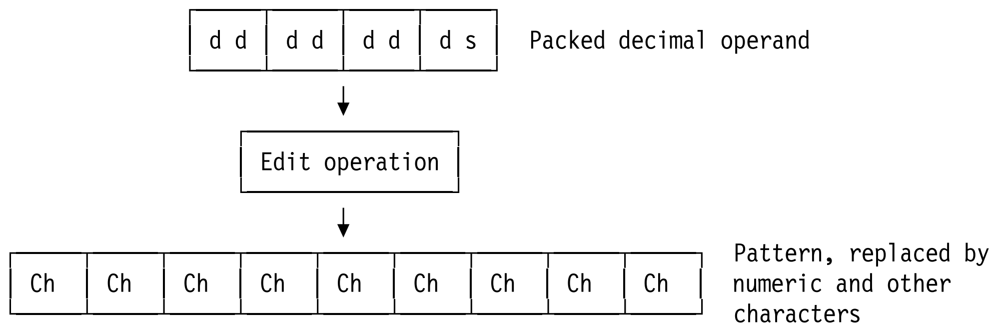


Figure 332. Sketch of an editing operation

[192] We'll see more about unsigned packed decimal data when we discuss Decimal Floating-Point in Section 35.

This picture is created from four types of information that tell the instruction how to convert the packed decimal second operand into zoned characters. Each byte in a pattern contains one of these five types of information:

1. a Fill Character (FC), the first byte of the pattern, which may have any value;
2. a Digit Selector (DS) character `X'20'`;
3. a Digit Selector and Significance Start (SS) character `X'21'`;
4. a Field Separator (FS) character `X'22'`;
5. a Message Character (MC), which may have any value other than `X'20'`, `X'21'`, or `X'22'`. It has no effect on the packed decimal second operand.

Each byte in a pattern is called a Pattern Character (PC).[193]

The editing process scans the pattern once from left to right, and takes action depending on

- which of the five kinds of pattern byte is encountered, and
- what has happened previously.

To remember what "happened previously", the CPU uses a single internal bit called the *Significance Indicator* (SI).[194]

The Significance Indicator is not a part of any register or of the PSW, and its value is not accessible to the program except as it influences the progress of an editing operation. It is set OFF at the start of an editing operation, and then it is turned on and off by events that occur during the edit. Its final setting may influence the Condition Code setting when the edit operation completes.

Visualizing patterns is difficult because the three non-message characters (with values `X'20'`, `X'21'`, and `X'22'`) have no printable representation. If we choose a printable character to represent each of them, then we might confuse them with ordinary message characters, because any printable character in a pattern is normally a message character!

We will use this notation: the lower-case letters `d`, `s`, and `f` represent respectively the Digit Selector (DS), Digit Selector and Significance Start (SS), and Field Separator (FS) characters. As before, we'll use "•" to represent a blank space.

Thus, we might represent the pattern

```
X'402020202120C3D922202120'   by   C'•dddsdCRfdsd'
```

Figure 333. Representation of an editing pattern

The first byte in a pattern is the *Fill Character* (FC). The CPU saves a copy of this first pattern character for the duration of the editing operation. It is used in various situations to replace other pattern characters. A common choice for a fill character is a blank, as in Figure 333.

To summarize the features described thus far:

1. The first operand is a pattern containing the Fill Character, and one or more Digit Selector, Digit Selector and Significance Start, Field Separator, and Message Characters.
2. The second operand is one or more packed decimal numbers to be converted to zoned format in a manner controlled by (and pictured by) the pattern.
3. The result replaces (overwrites) the pattern with Message Characters, Fill Characters, and zoned decimal digits. Thus the pattern is usually copied first to an editing area.

---

193 System z has had a "PC" since 1966.

194 In some older books and manuals, it was called the *S-Trigger*.

Exercises

30.3.1.(2)+ For each of the following patterns, identify (1) the Fill Character, (2) Digit Selectors, (3) Significance Start and Digit Selectors, (4) Field Separators, and (5) Message Characters.

1. X'40202020202120'
2. X'5C20202021204B2020'
3. X'5C4020202021202240E396A381937E20202021204B2020'
4. C' Hello, World!'

30.3.2.(2) An early programming convention represented the DS, SS, and FS characters by ', (, and) respectively. Rewrite the pattern of Figure 333 on page 537 using this convention, and assess its readability.

30.4. Simple Examples of Editing

We begin by illustrating the simplest forms of the editing process with a "plausible" example, and then give some more general rules. While the overall operation of ED and EDMK will turn out to be fairly straightforward, it may be difficult initially to understand why and how things are happening.

Suppose we are printing a report having titles and page numbers on each page, and we know there will be at most 900 pages in a report. This means that the two-byte packed decimal number at **PgNo** can hold the page number without overflow. We could use UNPK to produce a printable character string of the form "Page nnn" with these instructions.

```
         UNPK  TtlPgN(3),PgNo(2)  Convert to zoned format
         OI    TtlPgN+2,C'0'      Set correct zone on last digit
         - - -
PgNo     DC    PL2'7'             3-digit page number 007+
TtlP     DC    C'Page '           Start of page-number string
TtlPgN   DS    ZL3                Zoned page number
```

Figure 334. Convert a packed decimal integer to characters using UNPK

This has a defect: small page numbers will have leading zeros, as in "Page 007". We can use the ED instruction to both unpack the digits *and* suppress the leading zeros, as shown in Figure 335:

```
         MVC   TtlPgN(4),PgNPat   Move a copy of pattern
         ED    TtlPgN(4),PgNo     Convert to zoned format
         - - -
PgNo     DC    PL2'7'             Page number 007+
TtlP     DC    C'Page'            Start of page-number string
TtlPgN   DS    CL4                Edited result = C'•••7'
PgNPat   DC    C' ',3X'20'        Pattern = C' ddd'
```

Figure 335. Convert a packed decimal integer to characters using ED

Because we will want to print the page number more than once, we must use the MVC instruction to copy the unmodified pattern from **PgNPat** each time, where it will be replaced by the edited result.

The ED instruction converts the packed operand at **PgNo** as follows:

1. The first character of the pattern (a blank) is saved as the Fill Character, and the Significance Indicator is set OFF.
2. When the first Digit Selector character is encountered, the first digit of the second operand is examined: if it is zero (as in this case) *and* the Significance Indicator is OFF, the Fill Character replaces the pattern character.

3. The next pattern character and the next source digit are examined; because no significant (nonzero) digits have been encountered, the Fill Character again replaces the pattern character.
4. Finally, the last pattern character and second operand digit are examined. Because the latter is nonzero, it is converted to zoned format (X'F7'), and the zoned result replaces the pattern character. The Significance Indicator is set ON to show that a significant digit has been found.

The result C'•••7' contains three blanks: the first is the original Fill Character, and the other two are the suppressed leading zeros.

Now, suppose the page number at **PgNo** is 700+ instead of 007+. When the ED instruction is executed, the first digit examined is nonzero. The Significance Indicator will be set ON, and X'F7' will replace the first Digit Selector character. When the second and third pattern characters are examined, the status of the Significance Indicator shows that the zero digits from the second operand are now significant. They are therefore zoned, and replace the pattern characters. The result will be "Page 700" as desired.

From this simple example, we infer some simple rules about how editing works:

1. If the pattern character is a Digit Selector, the next decimal digit is taken from the second operand.
2. If the Significance Indicator is ON, go to step 5.
3. If the SI is OFF, examine the decimal digit. If it is *not* zero, set the SI ON and go to step 5.
4. If the SI is OFF and the decimal digit is zero, replace the pattern character (DS) by the Fill Character, and go to step 6.
5. Attach a zone to the decimal digit, and replace the (DS) pattern character with the result.
6. Step to the next pattern character and decimal digit, and repeat until the pattern is exhausted.

This description omits some important considerations, but it shows the basic features of the editing process. The number of decimal digits examined from the second operand is exactly the number of selector characters (DS and SS) in the pattern.

Suppose you now want to print any nonnegative 3-digit decimal number using the technique shown in Figure 335 on page 538. Everything works unless you try to print 000+: in the figure we assumed that page numbers start at 1, not at zero, so we didn't have to worry about this case. From the above description it is clear that using the pattern X'40202020' to edit 000+ gives an all-blank result! If we actually *wanted* to blank out a zero field, fine: we just discovered how to do it (by accident).

You will normally want to print at least one zero character in such cases. The "Digit Selector and Significance Start" (SS) character is used to force digit significance to begin.

The SS character works exactly like the DS character, except that it also turns the Significance Indicator ON if it was not already on. However, the Significance Indicator is set ON *after* the source digit has been examined, so significant digits will start (if they haven't already) in the *next* digit position! Thus, the SS character actually indicates not the start of significance, but more correctly the "end of insignificance", the rightmost limit of zero suppression.

Step 4 of the preceding "simple rules" can be revised as follows:

4'. If the SI is OFF and the decimal digit is zero, replace the pattern character (DS or SS) by the Fill Character. If the pattern character was SS, set the SI ON and go to step 6.

Suppose you need to print the value of the nonnegative binary integer fullword at **Num**. If the value is zero, a single 0 digit should be printed. Since a fullword integer can be at most 10 decimal digits long, we might plan to use an edit pattern with 10 digit selectors. With a fill character, the pattern would be 11 bytes long.

However, after converting the binary value to packed decimal, we find that the 8-byte decimal field (with 15 decimal digits) always contains at least 5 high-order zero digits that we want to

ignore. We can adjust the second operand address to skip four of the leading zeros, but because the digits taken by the editing instruction always start with the leftmost digit of the second operand, the pattern must contain 11 digit selectors to account for the remaining leading zero.

```
         L     0,Num               Get nonnegative number
         CVD   0,WorkArea          Convert to packed decimal
         MVC   LineX,Pat           Move pattern to print line
         ED    LineX,WorkArea+2    Start edit with high-order digits
         - - -
Num      DC    F'1234567890'       Number to be printed
WorkArea DS    D                   8-byte work area for CVD
Pat      DC    C' ',9X'20',X'2120'      Pattern = C'•dddddddddsd'
Line     DC    C' Num='            Start of printable text
LineX    DS    CL12                Edited result here
```

Figure 336. Converting a 32-bit binary integer to characters

The pattern at **Pat** contains 11 digit selector bytes; the next-to-last also starts significance, so the result will have at least one significant digit. The implied length (12) of the symbol **LineX** supplies the length byte in the MVC and ED instructions.

Exercises

30.4.1.(2)+ In Figure 336, what result will appear at **LineX** if the pattern at **Pat** contains only 10 digit selectors?

30.4.2.(2) Suppose a pattern is L bytes long. What is the maximum number of bytes that might be taken from the second operand? The minimum?

30.4.3.(2)+ Suppose we execute these instructions for each of the three indicated values of the pattern. What results will be generated?

```
         MVC   Output,Pattern
         ED    Output,Work8+2
         - - -
Work8    DC    X'000000000729413C'
Output   DS    CL12
```

The patterns are defined by

1. Pattern DC   C' ',11X'20'
2. Pattern DC   C'*',X'21',10X'20'
3. Pattern DC   C'*',3X'20',X'21',7X'20'

30.4.4.(2) In Exercise 30.4.3, what would have been generated if the second operand of the ED instruction was Work8+1 instead?

30.4.5.(1) What will result in Figure 336 if c(Num) = 0?

30.4.6.(2)+ Which DS characters in the pattern in Figure 336 will always be replaced by the Fill Character?

30.4.7.(2)+ What will happen if the length byte of an ED instruction contains 0 (the pattern is 1 byte long), and the pattern byte is not a DS or SS character, *and* the operand 2 address is invalid? What will happen if the pattern is longer than one byte but still contains no digit selectors?

30.5. Single-Field Editing

Now that we have a general idea of how editing works, we can give a more detailed description. We'll investigate the function of the Field Separator (FS) pattern character in Section 30.7.

Each step of an edit operation always gives one of *three* possible results:

1. a source digit from the second operand is zoned and stored into the pattern in place of a DS or SS character; or,
2. the Fill Character replaces the pattern character; or,
3. the pattern character is left unchanged.

Which of these three results occurs depends on the state of the Significance Indicator and the type of pattern character. If the pattern character is a Digit Selector of either type, the result also depends on whether the source digit is zero.

Message Characters in the pattern are either left unchanged (if the SI is ON) or replaced by the Fill Character (if the SI is OFF). They can be thought of as being “significant” or “not significant” Message Characters.

Suppose we want to print the nonnegative fullword integer at **Num** as in Figure 336 on page 540, and we want commas to separate every group of three digits (counting from the right). Thus, the binary integer 1234567890 should be edited to form the characters 1,234,567,890. This is done just as before, except that the pattern is now C'•dd,ddd,ddd,dsd'. (Remember that we use the • character to represent a blank.)

```
         L     0,Num              Get number to be printed
         CVD   0,WorkArea         Convert to packed decimal
         MVC   LineX,Pat          Move pattern to print line
         ED    LineX,WorkArea+2   Edit 11 decimal digits
         - - -
Num      DC    F'1234567890'      Number to be printed
WorkArea DS    0D,XL8             Work area for CVD
Pat      DC    C' ',3X'20206B20',X'2120'    X'6B' is a comma
Line     DC    C' Num ='          Start of printable text
LineX    DS    CL(Line-Pat)       Edited result here
```

Figure 337. Editing a binary integer with separating commas

The Message Character “comma” was written in its hexadecimal form, X'6B'.

Until a nonzero source digit (or the SS pattern character) is encountered, the SI will be OFF, and all pattern characters *including* the commas will be replaced by the Fill Character, a blank. Thus, if the value at **Num** is 4095, the first two commas in the pattern will be suppressed, leaving 4,095 as the nonblank part of the result.

30.5.1. Editing Negative Values

We have deferred considering the sign of the second operand, assuming that it was always non-negative. When the sign code at the end of the packed decimal operand is encountered, the Significance Indicator is set OFF if the sign code is +. This can be used to “fill” characters following the last digit.

For example, suppose you must print the balance in a charge-card account. If the balance is negative, indicating that the customer has a “credit” balance, you should print the word “CREDIT” following the amount. Let the packed decimal number at **Balance** represent the account balance in cents, so a decimal point must precede the last two digits of the edited result.

```
         MVC   LinB,Pat2          Move pattern to print line
         ED    LinB,Balance       Edit to printable form
         - - -
Balance  DC    P'-0012345'        Credit balance of $123.45
Pat2     DC    C' ',X'20206B2020214B2020',C' CREDIT'  Pattern
* Note: PAT2 = C'•dd,dds.dd•CREDIT'
PatX     Equ   *                  Used for defining length of Pat2
Line     DC    C' Your account balance is'
LinB     DS    CL(PatX-Pat2)      Space for edited result
```

Figure 338. Editing a signed number

When the sign code at the end of the second operand is encountered, the SI will be ON because nonzero digits were found in the second operand. Thus, the Message Characters following the last DS in the pattern are significant, and they will not be replaced by the Fill Character. Using "•" to represent a blank, the result would be

```
Your•Account•Balance•is••••123.45•CREDIT
```

Now, suppose the customer's balance is 0032109+, where the + sign code indicates he owes that amount: the word "CREDIT" should not be printed. In this case, the same instructions would produce

```
Your•Account•Balance•is••••321.09•••••••
```

Finally, suppose the customer's balance is 0000002−, indicating he has a credit of two cents. The SS character in the pattern immediately preceding the decimal point (X'4B') sets the SI ON, so the decimal point and the two following digits remain in the edited line. The printed result would then be

```
Your•Account•Balance•is•••••••.02•CREDIT
```

30.5.2. Protecting High-Order Fields

When printing negotiable items like paychecks, it is important to "protect" the high-order portion of the field. If we changed the contents of **Line** to read "Your pay is", an enterprising programmer could run his paycheck through his own printer and insert some extra "significant" digits to the left of the decimal point, perhaps to make it look like

```
Your•pay•is••$4,960.02•CREDIT
```

This is usually prevented by using a nonblank Fill Character such as * to protect the edited result.

```
         MVC   LPat,PayPat        Move pattern to line
         ED    LPat,PayAmt        Edit to protected print form
         - - -
PayAmt   DC    P'0098765'         Amount to print = 987.65
PayPat   DC    C'*',X'20206B2020214B2020'   C'*dd,dds.dd'
Line     DC    C'Dollars '        Start of Amount area
LPat     DS    CL(Line-PayPat)    Result = C'****987.65'
```

Figure 339. Using field protection with ED

and the printed result will be "Dollars•****987.65".

Exercises

30.5.1.(1)+ Which Message Characters set the Significance Indicator ON? Which ones set it OFF?

30.5.2.(1)+ In Figure 338, how would you modify the pattern to ensure that at least one digit appears before the decimal point?

30.5.3.(2) If a packed decimal number is P bytes long, how many Digit Selector and/or Significance Start characters should be in the edit pattern used to format it?

30.5.4.(4) Suppose a packed decimal operand is P bytes long, but you only want to edit N digits. Determine

1. the number of d and s selectors that must be in the pattern, and
2. the offset D from the start of the packed decimal operand that should be specified in the second operand of the ED instruction.

30.5.5.(2) What will result in Figure 337 on page 541 if c(Num) = 0? If c(Num) = 512, will there be a leading comma in the result? Explain.

30.5.6.(1)+ The edited result in Figure 337 on page 541 would be `C'••1,234,567,890'`. Explain why there are *two* blank characters.

30.5.7.(2)+ What would be the result in Figure 338 on page 542 if the value at **Balance** is 0+ instead?

30.5.8.(2)+ Do the same as in Exercise 30.5.7, but this time with the value at **Balance** now being 0−.

30.6. The EDMK Instruction

It is often useful to know where the most significant digit of an edited result occurs. For example, a minus sign customarily precedes negative results in integer arithmetic, and a currency symbol ($) might be placed immediately before the first significant digit of a dollar amount. The EDMK instruction *may* put the address of the first significant digit into general register 1, and then *only* if the SI is OFF when the first significant digit is stored into the pattern. If the first significant digit is stored after the first SS character in the pattern — that is, if significance is forced— register 1 will remain unchanged. (The flow diagram in Figure 346 on page 548 may help.)

The example in Figure 340 shows one way to insert a currency symbol, but it is incomplete:

```
        MVC   LPat,PayPat        Move pattern to Line
        EDMK  LPat,PayAmt        Edit and Mark result
        BCTR  1,0                Decrement GR1 (move left one byte)
        MVI   0(1),C'$'          Put $ sign before first digit
        - - -
PayAmt  DC    P'0098765'         Amount to print = $987.65
PayPat  DC    C' ',X'20206B2020214B2020'   C'•dd,dds.dd'
Line    DC    C'Pay Exactly'     Precedes the 'Amount:' area
LPat    DS    CL(Line-PayPat)    Result = C'•••$987.65'
```

Figure 340. Edited result with a floating currency symbol

This example will not work the way we want if the packed decimal number at **PayAmt** is 99 or less, because the pattern contains a Significance Start character just before the decimal point which will force significance of the last three pattern characters. Because GR1 would then be unchanged by the EDMK, we must preset its contents to the address of the byte *following* the Significance Start pattern character, which will be the first significant result character if significance is forced.

```
         MVC   LPat,PayPat       Move pattern to print area
         LA    1,LPat+L'LPat-3   Point to 1st significant pat'n char
         EDMK  LPat,PayAmt       Edit and mark result
         AHI   1,-1              Move to left of 1st significant char
         MVI   0(1),C'$'         Put the $ in the result
         - - -
PayAmt   DC    P'0000002'        Two cents pay for all this work?
PayPat   DC    X'4020206B2020214B2020'   Same pattern
Line     DC    C'Pay Exactly'    Start of print line
LPat     DS    CL(L'PayPat)    Result = C'••••••$.02'
```

Figure 341. Edited result with a properly placed floating currency symbol

If we want to print edited integer results with leading signs, we must know which of the "+" and "−" characters to place before the first significant digit. This can be determined by testing the Condition Code at the end of the edit:

CC	Meaning
0	All source digits 0, or no digit selectors in pattern
1	Nonzero source digits, and SI is ON (result < 0)
2	Nonzero source digits, and SI is OFF (result > 0)

Table 219. CC settings after ED, EDMK

The CC can be relied on to give a valid sign indication only if the last source byte taken from the second operand contained a sign code in its right digit.

Suppose we now revise Figure 337 on page 541 to allow values of either sign, with the added requirement that zero values are to be printed with no sign character. (The PrintLin macro is described in "Appendix B: Simple I/O Macros" on page 1015.)

```
         L     1,Num             Get number to be printed
         CVD   1,WorkArea        Convert to packed decimal
         MVC   LineX,Pattern     Move pattern to print line
         LA    1,LineX+L'LineX-1 Point to possibly forced digit
         EDMK  LineX,WorkArea+2  Edit and mark up to 11 digits
         JZ    Print             If zero, print immediately
         BCTR  1,0               Adjust to preceding byte
         MVI   0(1),C'+'         Assume result was +
         JP    Print             Branch if the guess was right
         MVI   0(1),C'-'         Negative result, set - sign
Print    PrintLin   Line,(L'Line+L'LineX)   Print result
         - - -
Num      DC    F'-1234567890'    Number to be printed
WorkArea DS    D                 Doubleword work area for cvd
Pattern  DC    C' ',3X'20206B20',X'2120'    Pattern
Line     DC    C' Num ='         Start of printed output
LineX    DS    CL(Line-Pattern)  Result = C'•-1,234,567,890'
```

Figure 342. Integer value with optional sign and separating commas

We use BCTR 1,0 here instead of AHI 1,−1 to decrement the address in GR1 because AHI changes the Condition Code.

Exercises

30.6.1.(3) Packed decimal numbers can represent minus zero, but the CC setting after an EDMK instruction only indicates that all source digits are zero. Write an instruction sequence using EDMK to display a correctly signed 4-byte packed decimal number at **PackVal** even if its value is zero.

30.6.2.(2)+ What CC setting will result from editing the packed decimal operand 000− with the pattern X'C1212020'? What will be the result?

30.7. Editing Multiple Fields (*)

More than one edited result can be produced by a single execution of an ED or EDMK instruction. The fields of the result are separated in the pattern by the Field Separator (FS) character, X'22'. The FS character (a) resets the zero-digits test that will be used to determine the final CC setting (essentially, it resets the CC to 0), (b) sets the SI OFF, and (c) is replaced by the Fill Character. Thus, each field of the pattern can be considered separately from the others; however, only the last field can be "marked" by an EDMK instruction, since the address in general register 1 will refer to the last significant digit stored while the SI was OFF.

For example, suppose we want to edit two packed decimal values into a single field:

```
         ED    Pat2,PD2            Edit two packed decimal values
         - - -
PD2      DC    P'+024',P'-135'     Two values
Pat2     DC    X'402021204022202120'   C'•dsdf•dsd'
```

Figure 343. Editing two packed decimal numbers into a single field

and the edited result is C'••24••123'.

As a more complex example, suppose there are three packed decimal numbers in successive fields named **Hours**, **Rate**, and **PayAmt** to be edited into a single print line.

```
         MVC   LPat,Pat            Move pattern to line
         ED    LPat,Hours          Edit all 3 fields at once
         - - -
Hours    DC    PL2'35.5'           Hours worked, in tenths
Rate     DC    PL3'7.50'           Pay rate in dollars/hour
PayAmt   DC    PL4'266.25'         Pay amount = $266.25
Pat      DC    C' ',X'20214B20'            C'•ds.d'
         DC    X'22',X'2020214B2020'       C'fdds.dd'
         DC    X'22',X'20202020214B2020'   C'fdddds.dd'
Line     DC    C' Hours  Rate       Pay'    title line
LPat     DS    CL(Line-Pat)        Space for edited pattern
```

Figure 344. Editing multiple values

The two lines would then contain

```
 Hours  Rate      Pay
 35.5   7.50   266.75
```

While we would like to use EDMK here to insert a "$" character before the first significant digit of the "Pay" field of the result, this may not work. Suppose the amount to be paid is 99 cents or less. Then the address in general register 1 would remain at the address of the first significant digit of the last field in which significance was *not* forced, which would be the "3" in the "Rate" field of the result!

Even if we had preset register 1 to the address of the decimal point in the last field of the pattern, we could still find ourselves placing the $ sign in the wrong field. Thus, it is rare to use EDMK in editing multiple fields with floating currency symbols unless you can guarantee that no such problems can arise.

Exercises

30.7.1.(2)+ Suppose a pattern contains N digit selectors. What are the maximum and minimum number of bytes that might be taken from the second operand?

30.7.2.(3) Suppose a Field Separator character is encountered in an edit pattern when the next second operand digit to be examined is in the right half of a source byte. Will the next digit selector in the pattern after the FS cause that source digit to be examined, or will the FS pattern character cause the next source digit to come from the left half of the following second operand byte? (You may want to consult the *z/Architecture Principles of Operation*!)

30.7.3.(2)+ Suppose you use EDMK to edit a multiple-field pattern. Under what conditions can you assume that GR1 refers to a character in the last field only?

30.7.4.(2)+ In Figure 344 on page 545, what changes would happen to the output if we had defined **Hours** with this value instead?

```
Hours    DC    PL2'5'
```

30.7.5.(2) Repeat exercise 30.7.4, but with these definitions:

```
Hours    DC    PL2'1'
Hours    DC    PL2'0.4'
```

30.7.6.(3)+ Repeat Exercises 30.7.4 and 30.7.5, this time with the edit pattern named by the symbol **Pat** now containing this pattern?

```
Pat      DC    C' ',X'20204B20'
```

30.7.7.(3)+ In Figure 343 on page 545, what edited result would be created with the same pattern but with packed decimal arguments defined by:

```
(1)  Pack2   DC    P'7',P'2468'
(2)  Pack2   DC    X'67891234'
```

30.7.8.(2) Suppose you edit the unsigned packed decimal value `X'0123456789'` using the multiple-field pattern `C'•sss•fss•fsssss'`. What will be the result?

30.8. Summary Comments on Editing (*)

Before we give a complete summary of the actions of the ED and EDMK instructions, there are two minor items to consider:

1. In the source bytes brought from the second operand, only valid decimal digits from 0 to 9 may appear in the left digit of a source byte. If a sign code appears in the left half of a source byte, a program interruption will occur, and the Interruption Code will be set to 7. The contents of the result field (and the contents of general register 1 if the instruction is EDMK) is unpredictable.
2. There is no point in trying to overlap the first and second operand fields in an editing operation, because the results are quite unpredictable.

We will summarize the action of the editing instructions in three different ways:

1. Table 220 on page 547 shows what happens for each of the four different types of pattern characters.
2. In a set of "logical" relations in Figure 345 on page 547.
3. In a flow diagram in Figure 346 on page 548.

The abbreviations used are the same as previously introduced, now including ZD (Zoned Digit) and PC (Pattern Character).

The actions caused by each of the four types of pattern character is described in Table 220 on page 547, where SI=0 means OFF and SI=1 means ON.

Pattern Character (PC)	**Get Source Digit?**	**SI**	**Source Digit**	**Result**	**Set SI**	**Sign code in right digit?**
X'20' (DS)	Yes	1	Any	ZD	1	If +, set SI OFF; otherwise, leave it unchanged
		0	Nonzero	ZD	1	
		0	Zero	Fill Char	0	
X'21' (SS)	Yes	1	Any	ZD	1	If +, set SI OFF; otherwise, leave it unchanged
		0	Nonzero	ZD	1	
		0	Zero	Fill Char	1	
X'22' (FS)	No	—	—	Fill Char	0	No source byte is examined
Other (MC)	No	0	—	Fill Char	0	No source byte is examined
		1	—	MC	1	
Fill Character (FC)	No	N/A	—	Fill Char	0	N/A

Table 220. ED and EDMK treatment of pattern characters

Using the following logical symbols to represent logical operators:

```
& → AND,    | → OR,    ¬ → NOT
```

we can write a simple set of “logical equations” that describe the action of the editing process in this compact form, where SD means “Source Digit” from the second operand, and ZD means “Zoned Digit”.

```
                  SI & MC ──→ MC

           FS | ((¬SI) & MC) ──→ FC

     (DS | SS) & SD=0 & ¬SI ──→ FC        (insignificant zero)

    ((DS | SS) & SI) | SD≠0 ──→ ZD, SI=1  (significant digit)
```

Figure 345. Logical-operation description of the editing process

This compact notation does not describe the setting and resetting of the Significance Indicator, but those actions were discussed in detail earlier.

A flow diagram of the editing process is given in Figure 346 on page 548.

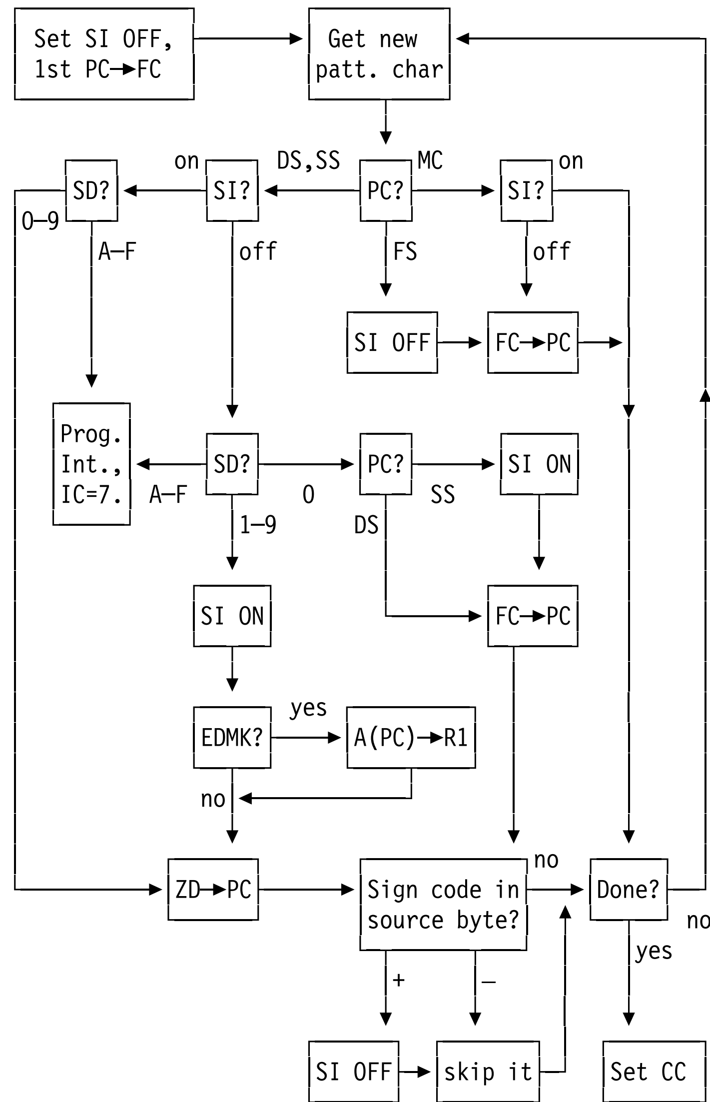


Figure 346. ED and EDMK operation

Exercises

30.8.1.(2)+ Prepare a short summary showing when the Status Indicator is set ON and OFF.

Terms and Definitions

digit selector
: One of two edit-pattern characters: a Digit Selector (DS) having representation X'20', or a Digit Selector and Significance Starter (SS) having representation X'21'.

field separator
: An edit pattern character (FS) having value X'22' indicating that a packed decimal value from the second operand has been edited, and editing should continue with the following packed decimal value.

fill character
: The first byte of an edit pattern.

message character
Any character in an edit pattern that is not a Digit Selector (DS), a Field Separator (FS), or a Digit Selector and Significance Starter (SS).

pattern character
Any byte in an edit pattern.

significance indicator
An internal bit used by the CPU during ED and EDMK instructions to control subsequent editing operations.

significance starter
An edit- pattern character (SS) having value `X'21'` that sets the Significance Indicator ON. Also known as a "digit selector and significance starter" because it selects a digit if the Significance Indicator is already on.

zoned digit
An unpacked or edited packed decimal numeric digit.

Programming Problems

Problem 30.1.(3) Suppose you are writing a small Assembler in which space is very limited. Your Assembler must store and manipulate 6-character alphanumeric symbols in a compact representation that packs the symbol into a fullword, as follows:

1. The leftmost 12 bits of the fullword contain six pairs of "zone" bits. Each character in the symbol is a letter or digit, its EBCDIC representation must lie between `X'C1'` and `X'F9'`. Because the two leftmost bits are always `B'11'`, we take the *next two* bits for the "zone" portion of the character.
2. The rightmost 20 bits of the fullword contain the *numeric* portions of the six characters, expressed as a *binary* number. (Since $2^{20} > 10^6$, there is enough room to contain the six numerics.)

   For example, the symbol A2N6Z8 (=`X'C1F2D5F6E9F8'`) would be encoded as 2^{20}×`B'001101111011'` + 125698, or `X'37B1EB02'`.
3. Since symbols may be shorter than 6 characters in length, we can let the combination `B'00'+0 (X'C0')` represent a blank.

Write a program that reads valid symbols from records to produce the fullword "packing" just described. Then convert the "packed" format back to characters, and display the original symbols and the unpacked result. What collating sequence is defined by this packing method?

Problem 30.2.(4) Repeat Problem 24.2 (the "perfect shuffle" program), using packed decimal data to represent the card values, and an ED or EDMK instruction to format the output.

Problem 30.3.(2)+ You can convert a string of numeric characters to binary by multiplying and adding. For example, the characters 345 can be converted to binary by evaluating ((3×10)+4)×10)+5. Write a program that reads records containing a string of decimal characters, and convert them to 32-bit binary. Display the original characters and the converted results in any convenient way to verify your conversion. If any value represented by the numeric characters won't fit in 32 bits, display the word "`Overflow`" instead of the converted value.

Problem 30.4.(3)+ Revise your solution to Problem 18.6 to read records containing a Year as a string of decimal digits. If the year lies outside the range (1583,9999), print an error message and read the next input record. Otherwise, calculate the date of Easter, and print the result in a format like this:

```
Easter in year 2010 is on Sunday April  4.
```

Problem 30.5.(3)+ Write a program to read records containing one or more unsigned integers in character form, separated by one or more blanks. Display the value of each in character, decimal, hexadecimal, and binary formats in aligned columns. Print appropriate messages if an input string is invalid.

Problem 30.6.(2)+ Write a program to read records containing an optionally signed integer in character form, and print a message stating whether or not its value is a power of two.

Problem 30.7.(2)+ Write a program to calculate the Fibonacci numbers, defined by the relations

```
F(N) = F(N-1) + F(N-2)
```

up to the largest value representable as a 32-bit binary number. Start with initial values `F(0)=0` and `F(1)=1`, so that the values will be `F(1)=1, F(2)=1, F(3)=2`, etc. Display them as decimal characters without leading zeros, and without using either ED or EDMK; that will be in Problem 30.8. (This is a variation on Problem 16.8.)

Problem 30.8.(3)+ Write a program to calculate the Fibonacci numbers, defined by the relations

```
F(N) = F(N-1) + F(N-2)
```

up to the largest value representable as a 32-bit binary number. Using an ED instruction to format the results, print the values with commas inserted between every three digits, as in `1,234,567`. Start with initial values `F(0)=0` and `F(1)=1`, so that the values displayed will be `F(1)=1, F(2)=1, F(3)=2`, etc.. (This is a variation on Problems 17.4 and 30.7.)

Problem 30.9.(3)+ Suppose you are given a set of data records that contain a name in columns 1-20, and an amount (in cents) in columns 21-28 (that is, there is an assumed decimal point between columns 26 and 27). Write a program which will read any number of such data records, and produce as output a list of totals, names, and numbers of records with that name. Thus, the input data

```
Smith                 1798
Jones                10000
Smith                  125
```

would produce output something like

```
Name                 Total    Items

Smith                19.23        2
Jones               100.00        1
```

Your program should be able to read any number of sets of data records; devise some means of separating them from one another.

Chapter IX: Floating-Point Data and Operations

```
IIIIIIIIII  XX        XX
IIIIIIIIII  XX        XX
    II       XX      XX
    II        XX    XX
    II         XX  XX
    II          XXXX
    II          XXXX
    II         XX  XX
    II        XX    XX
    II       XX      XX
IIIIIIIIII  XX        XX
IIIIIIIIII  XX        XX
```

The sections of this chapter discuss floating-point number representations and their uses.

- Section 31 introduces the basic concepts of scaled fixed-point and floating-point data, including the hexadecimal, binary, and decimal floating-point data types supported by System z. Then it describes the floating-point registers and some basic instructions for moving floating-point operands.
- Section 32 describes some general principles of floating-point arithmetic.
- Section 33 examines the hexadecimal floating-point representation of numbers, and describes hexadecimal arithmetic and instructions.
- Section 34 discusses the binary floating-point representation and instructions for doing binary floating-point arithmetic.
- Section 35 discusses the decimal floating-point representation and instructions for doing decimal floating-point arithmetic.
- Section 36 summarizes the three System z floating-point representations, and describes some important differences between mathematical “real” arithmetic and floating-point arithmetic.

31. Floating-Point Numbers: Introduction

```
 3333333333        11
333333333333      111
33        33     1111
          33       11
          33       11
       3333        11
       3333        11
          33       11
          33       11
33        33       11
333333333333   1111111111
 3333333333    1111111111
```

Why was floating-point arithmetic invented? How did programmers handle data with both integer and fraction parts when only fixed-point binary arithmetic was available? Before we investigate floating-point numbers and arithmetic, a bit of background may help.

31.1. Scaled Fixed-Point Arithmetic

Up to now, most of our examples involving binary arithmetic have used mostly unscaled integer values, as illustrated in Figure 347.

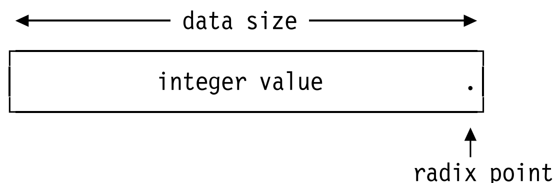


Figure 347. A data item containing an integer value

As we saw in Section 29.10 on page 522, packed decimal arithmetic may frequently need to manipulate values with both integer and fraction parts, as illustrated in Figure 348, where the decimal point is implied (not actually present).

Similarly, with fixed-point binary instructions, you can use numbers with *fractional* parts by assuming that a number is part integer and part fraction, with the position of the implicit radix point defined to be somewhere “inside” the number, as in Figure 348.

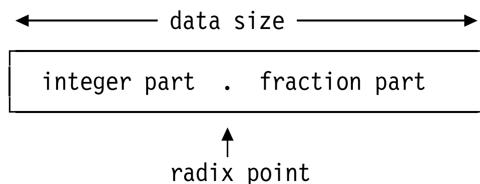


Figure 348. A data item containing integer and fraction parts

We call these quantities *fixed-point* data. You can write instructions that take account of this *scaling* (the position of the radix point) when doing arithmetic with such quantities. For example, when we process financial data, amounts are often stated in dollars and cents, Euros and cents, and the like: the assumed decimal point lies to the left of the two rightmost decimal digits.

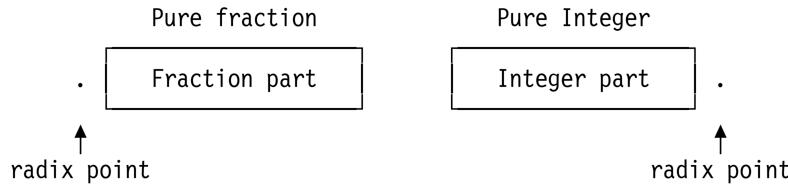


Figure 349. Values with radix point outside the digits

31.2. Mixed Integer-Fraction Representation

Early computers provided only integer representations and arithmetic, so values with integer and fraction parts were approximated by scaled fixed-point numbers. Programmers had to keep track of the radix point for each variable, and programmed shifts or other corrections for re-scaling after each operation.

For example, suppose an importer has received a shipment valued at $11749.49, and import tax must be paid at a rate of 5.147 percent. Your job is to calculate the amount of tax to be paid. (Remember, government auditors will be verifying your work!)

First, you calculate the product of the two numbers:

```
$11749.49 × 5.147% = $604.7462503
```

But the actual calculation using fixed binary or packed decimal arithmetic would be

```
1174949 × 5147 = 6047462503
```

Knowing that there are two fraction digits in $11749.49 and five fraction digits in 0.05147 (the percent rate), you know that there will be seven fraction digits in the product:

```
 $604.7462503
```

You would then round this to the nearest cent:

```
 $604.7462503
+       5
 $604.75
```

This is fairly easy to program on a processor with instructions that do decimal arithmetic. For example, you can multiply, shift, and round with just a few packed decimal instructions, as in Figure 350.

```
         ZAP   PDProd,PDAmt       Copy amount to work area
         MP    PDProd,PDTax       Multiply by tax rate
         SRP   PDProd,64-5,5      Convert to cents and round
         - - -
PDAmt    DC    P'11749.49'        Amount to be taxed
PDTax    DC    P'.05147'          Tax rate
PDProd   DS    PL8                Resulting product
```

Figure 350. Calculating a tax amount in scaled fixed decimal arithmetic

The result can then be formatted for printing. (See Programming Problem 31.1.)

But what will you do if your CPU supports only fixed binary arithmetic? This difficulty faced many early programmers.

Our little tax example is relatively easy to solve with fixed-point binary arithmetic. We treat all quantities as integers (noting that the product is more than 31 bits long), and remember how many powers of 10 to divide by to give a quotient that is fewer than 32 bits long. Thus, we could use binary arithmetic as in Figure 351.

```
         L     1,Amount           Put amount in cents in GR1
         M     0,Percent          Multiply by integer-ized percent
         AL    1,Round            Add a rounding factor
         JC    12,NoCarry         Skip if no carry occurred
         AHI   0,1                Propagate the carry bit
NoCarry  D     0,Correct          Divide by some power of 10
         - - -
Amount   DC    F'...'             Cost of shipment
PerCent  DC    F'...'             Percentage to multiply by
Round    DC    F'...'             Rounding factor
Correct  DC    F'...'             Correction for scaling
```

Figure 351. Calculating a tax amount in scaled fixed binary arithmetic

The quotient in GR1 can then be converted to decimal and formatted.

The actual values of the four constants are for you to determine: first, solve Exercise 31.2.1 and then do Programming Problem 31.2.

31.2.1. Scaled Fixed-Point Binary Arithmetic (*)

Obscure topic

Scaled fixed-point binary arithmetic is rarely used today; it has occasional application to calculating time differences using the CPU clock.

The import-tax example above is fairly straightforward. A common problem with mixed integer-fraction data is choosing the scaling to maximize precision while retaining the maximum range of represented values. When we must do arithmetic with scaled numbers of the form **Int.Frac** (with FD fraction digits), we need to know two things:

1. the largest magnitude of **Int**, and
2. the number of bits needed to represent **Frac** to FD significant decimal digits.

The number of bits needed to represent **Int** is

$\log_2(\text{Int})+1$

and the number of bits needed for **Frac** is

$\log_2(\text{FD})+1$

where the logarithm values have been truncated to an integer, which accounts for the +1 in each case. Reserving an additional single bit for the sign, the word length used to represent a signed value like **Int.Frac** must satisfy

$\text{word length} \geq \log_2(\text{Int})+\log_2(\text{FD})+3$

This inequality sometimes limits the range or precision of values that can be used in fixed-point arithmetic.

Now, consider a more difficult problem: suppose you must calculate the square root of 2 (1.414213562...) using fixed binary or packed decimal arithmetic, to 8 significant decimal digits (1 integer and 7 fraction digits). This can be done using a “Newton-Raphson” iteration. Let A be the number whose square root x is desired; if the initial estimate for x is reasonable, the value of x' will converge rapidly to the square root of A:

$$x' = x + (A/x - x)/2$$

We can use 1 as our initial estimate.

Because there is only a single digit to the left of the decimal point, we can use a binary representation with just a few bits to the left of the binary point, leaving the remaining bits to represent the fraction. For example, if the leftmost four bits of a word are the integer part, the remaining 28 bits will hold the fraction. The number 2 in this representation will be X'20000000', and the initial estimate 1 will be X'10000000'. (See Programming Problem 31.3.)

A problem like finding the square root of a small number like 2 is relatively straightforward, because all quantities have the same scaling. More often, the operands have different scale factors, so it takes more effort — as we saw for packed decimal arithmetic — to manage the operands while preserving enough precision in the intermediate and final results.

The need to manage problems like this led to the development of floating-point data and arithmetic, which maximizes precision and manages scaling automatically.

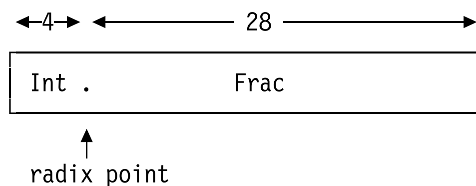
31.2.2. Scaled Fixed-Point Binary Constants (*)

The Assembler can do some scaling for you. For example, the values of **A** and **x** in the Newton-Raphson iteration can be written as *scaled* fixed-point constants:

```
A        DC    FS(28)'2'          2, scaled by 2**28     X'20000000'
X        DC    FS(28)'1'          1, scaled by 2**28     X'10000000'
```

Figure 352. Two binary constants scaled by 2**28

where the letter S following the constant type asks the Assembler to scale the result by shifting the nominal value to the left 28 bits. Other values used in the calculation are then scaled accordingly. In effect, we are defining two constants of this form:



Suppose we want to do scaled binary arithmetic with numbers having integer parts between 0 and 1000, which will need about 10 bits. And because we might add two numbers near 1000, we'll allocate 11 bits for the integer value and one bit for a sign. This leaves 20 bits for the fraction part.

How should the constant +123.4567 be represented?

1. The integer part of the constant is 123, or X'07B', so we know that the first twelve bits of the constant will be X'07B'. (The sign bit is 0 for +.)
2. Following the steps in Section 31.3 for converting fractions, we find that 0.4567_{10} = X'74EA4A8C...'. Because only 20 bits are allocated to the fraction, we must round this to five hex digits, giving X'74EA5'
3. Combining the integer and fraction parts, we find that the constant is X'07B74EA5'.

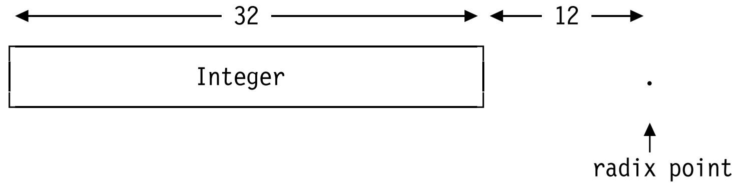
The main problem is keeping track of the scaling of intermediate quantities during the iterations, and formatting the decimal digits of the result's fraction part. (See Programming Problem 31.3.)

In addition to scaled fixed-point binary constants like those in Figure 352, you can also define scaled constants where the radix point lies *outside* the constant! For example, suppose you want to represent 10^{12} in a 32-bit word. But 10^{12} is greater than 2^{31}, so it might seem impossible. But because there are 12 low-order zero bits in 10^{12}, we can ask the Assembler to create a constant without those 12 zero bits; we must then remember to compensate for them when doing arithmetic with the constant:

```
TenTo12  DC    FS(-12)'1E12' 10**12 scaled by -12 bits, X'0E8D4A51'
```

Figure 353. Defining a scaled binary constant 10**12

where the negative scale factor −12 means that the Assembler will shift the binary constant to the right by 12 bits before generating the machine language constant. Effectively, you will generate a constant of this form:



If we want to multiply the scaled constant 2 in Figure 352 on page 555 by the scaled constant 10^{12} in Figure 353 on page 555, we can use instruction like these:

```
         L     1,A                c(GR1) = 2×2**28
         M     0,TenTo12          Multiply by 10**12×2**(-12)
         SRDA  0,16               Compensate for both scale factors
         - - -
TenTo12  DC    FS(-12)'1E12'      10**12 scaled by 12 bits
```

Figure 354. Multiplying two scaled binary numbers

and the 64-bit result in c(GR0,GR1) will be 2×10**12, or `X'000001D1 A94A2000'`. If this value will be used in later arithmetic, it will once again need to be scaled by 2**(−12) so that it will fit in a single register: `X'1D1A94A2'`.

Programming on processors with only fixed-point arithmetic isn't easy!

Exercises

31.2.1.(2) Write DC statements to define the values named **Amount**, **Percent**, **Round**, and **Correct** in Figure 351 on page 554.

31.2.2.(2) What machine language constant will be generated by the DC statement in Figure 353 on page 555?

31.2.3.(1) What machine language constants will be generated for the two packed decimal constants in Figure 350 on page 553, and how long must the work area named **PDProd** be?

31.2.4.(3) Convert these decimal values to hexadecimal, giving at most six significant fraction digits:

1. 6.3725
2. 987.603
3. 314.1592654

31.2.5.(1) With signed 32-bit words, how many different integer-fraction representations can be created? With unsigned 32-bit words?

31.2.6.(2) Referring to the import-tax example in "31.3. Converting Fractions Between Bases (*)" on page 557, what is the minimum number of bits needed to represent the integer portion of the signed quantity +604.7462503?

31.2.7.(2) Given π=3.14159265359, what is its hexadecimal representation as a 32-bit scaled fixed-point binary number with 28 fraction bits?

31.2.8.(4) The binary representation of each power of 10 has as many trailing zero bits as the power. For example, 10^2 = `X'64'` = `B'1100100'`.

Determine the largest *scaled* power of 10 that can be held in a 32-bit and a 64-bit signed two's complement constant. Then, determine the largest *unsigned* values for both lengths.

31.2.9.(2)+ What would happen if you had written the constant **TenTo12** in Figure 354 this way?

```
TenTo12  DC    FS(-12)'10E12'      10**12 scaled by 12 bits
```

31.2.10.(3)+ Given a scaled binary constant at **Value** representing the decimal value 98.234567, convert its value to EBCDIC characters with 5 digits after the decimal point, and with the last digit rounded.

31.3. Converting Fractions Between Bases (*)

In “2.3. Converting Integers from One Base to Another (*)” on page 19 we converted integers from one base to another by successive divisions, generating digits in the new base from least to most significant. For fractions, we do the reverse, using successive *multiplications* to generate digits from most to least significant.

Let the string of digits

$$0\ .\ d_1\ d_2\ d_3\ \ldots\ d_n$$

be the representation in some base D of the fraction X: that is,

$$X = \mathrm{SUM}(k=1 \text{ to } n)\ (d_k \times D^{-k}) = d_1 \times D^{-1} + d_2 \times D^{-2} + \ldots + d_n \times D^{-n}.$$

For example, the decimal fraction 0.1234 is

$$0\ .\ 1\times10^{-1} + 2\times10^{-2} + 3\times10^{-3} + 4\times10^{-4}$$

Now, suppose we want to convert X from its known representation in base D to its representation in a new base B: that is,

$$X = \mathrm{SUM}(k=1 \text{ to } m)\ (b_k \times B^{-k}) = b_1 \times B^{-1} + b_2 \times B^{-2} + \ldots + b_m \times B^{-m}.$$

We know the old and new bases D and B, and the digits d_k of the old representation. To find the digits b_k in the new representation, we do the following:

1. Multiply X by the new base B. Save the fraction part of the result, and the integer part is b_1. This can be seen by writing the product as

   $$B \times X = b_1 + [\ b_2 \times B^{-1} + b_3 \times B^{-2} + \ldots + b_m \times B^{(-m+1)}\ ]$$

2. Multiply the saved fraction part (the term in square brackets) by B again, and save the new fraction part. The generated integer part will be the second digit b_2.
3. Repeat the process until you have generated as many digits as you need, or until the saved fraction part is zero. Note that the digits are produced in order of *decreasing* significance.

There is no automatic terminating condition for fractions as for integers, because there are finite-length fractions in one base which do not have finite representations in another; the decimal representation of the fraction 1/3 is a well-known example. But if the saved fraction part becomes zero, the conversion terminates and the representation in the new base has a finite number of digits. Here are some examples.

1. Give the base two representation of the decimal fraction 0.375.

.375	.75	.5
× 2	× 2	× 2
0.750	1.50	1.0
b_1=0	b_2=1	b_3=1

   Since the fraction part after three multiplications is zero, the binary representation of 0.375 terminates, and we know that 0.375 (base 10) = 0.011 (base 2).

2. Give the base 16 representation of the decimal fraction 0.1.

```
 0.1     0.6     0.6
×16     ×16     ×16
 1.6     9.6     9.6
b1=1    b2=9    b3=9
```

In this case, repeated multiplication will continue to generate “9” digits, and the base 16 representation will not terminate. We can therefore write

0.1 (base 10) = 0.19999... (base 16) = 0.1<u>9</u> (base 16) = X'0.1<u>9</u>',

where the underscore indicates that the underscored digit (or group of digits) should be repeated indefinitely.[195] Thus, 2/3 (base 10) = 0.<u>6</u> (base 10), 0.1 (base 3) = 0.<u>3</u> (base 10), and 0.1 (base 10) = 0.0<u>0011</u> (base 2).

When we represent such numbers in a computer we must settle on enough digits to represent the number. This means it is very likely there will be some necessary inaccuracy in the result.

3. Give the decimal representation of X'0.FA'. First, we note that

   X'0.FA' = 15/16+10/256 = 250/256 = 0.9765625

   Now, we do the conversion arithmetic in base 16; if we do it in decimal we must convert the fraction to base 10 first, which is the object of the conversion!

```
0.FA    0.C4    0.A8    0.9     0.A     0.4     0.8
×  A    ×  A    ×  A    ×  A    ×  A    ×  A    ×  A
9.C4    7.A8    6.90    5.A     6.4     2.8     5.0
b1=9    b2=7    b3=6    b4=5    b5=6    b6=2    b7=5
```

   so that 0.FA (base 16) = 0.9765625 (base 10), as expected. Converting between decimal and hexadecimal fractions like these can sometimes be done using the tables in the Appendix at “Conversion Tables for Hexadecimal Fractions” on page 1011.

Exercises

31.3.1.(3)+ Perform the indicated conversions. For number bases greater than 10, use the hexadecimal “digits” A, B, C... corresponding to 10, 11, 12....

1. Convert 0.1 (base 10) to bases 2 and 16.
2. Convert 0.3142 (base 10) to bases 3 and 14.
3. Convert 0.BBBBB... (base 16) to bases 10 and 15.
4. Convert 3.6 (base 8) to bases 10 and 16.

31.3.2.(2)+ Convert the following fractions from base 16 to base 10, giving the result to at most 7 decimal digits:

1. X'0.DEFACE'
2. X'0.5'
3. X'0.C2854'
4. X'0.333333'
5. X'0.BEEF'

31.3.3.(2) Convert these decimal fractions to hexadecimal, giving at most six significant digits:

1. 0.063725
2. 0.0001
3. 0.987603
4. 0.000005

31.3.4.(1) What is the binary representation of the fraction 1/3?

[195] The more common mathematical notation for repeated groups of digits is an over-bar (called a *vinculum*), but the software formatting these notes insists on unnatural acts to print over-bars.

31.3.5.(2)+ Convert the decimal fractions 0.1 through 0.9 in increments of 0.1 to base-8 (octal) format.

31.4. Why Use Floating-Point Numbers?

Floating-point arithmetic originated primarily to satisfy the needs of scientific cimputation. It sometimes seems unusual (or even frightening) compared to fixed-point arithmetic. It is also sometimes thought to be prone to errors; but it is no different in that respect from fixed-point arithmetic — and sometimes, better. The examples of computing import tax in Figures 350 and 351 on page 554 actually introduced errors, because the results had to be rounded and presented to less than full precision.

But you *will* want to be careful. Most floating-point computations are reliable, simple, fast, and accurate, while others can give misleading (or even wrong!) answers.

Because actual data can take many magnitudes, the floating-point number representation helps us manage data of widely varying values. Figure 355 has examples of various constants:

```
    6022 00000 00000 00000 00000.                Avogadro's Number
             9 46000 00000 00000.                 meters/light-year
                      2997 76000.                  meters/sec (light)
                           33136.                   cm/sec (sound)
                             745.7                   Watts/horsepower
                             440.                     hertz (Concert A)
                              16.3872                  cc/cubic inch
                               3.14159 26535 8979       π
 kilometers/mile               1.60935
  ln 2                          .69314 71805 59945
   Coulombs/electron            .00000 00000 00000 00016
    Planck's const (Joule-sec)   .00000 00000 00000 00000 00000 00000 00066 26
```

Figure 355. Examples of data with widely ranging values

Managing these widely varying magnitudes with fixed point binary arithmetic is rather difficult. Like most results in fixed-point arithmetic, we are mainly concerned with the most significant digits, and we use a power of some base such as 2, 10, or 16 by which those digits should be multiplied.

Thus, floating-point data and arithmetic have many advantages:

- Because the most significant digits are retained, values have more uniform precision. Scaled fixed-point representations may need to use less precision for the fraction part of a number so they can correctly represent the largest values of their integer part.
- Our scaled fixed-point examples used 32-bit fullwords, but they were very limited in the range of values they could represent. Floating-point variables give up some bits of precision in order to support a much greater range.
- Precision losses involve the *least* significant digits, which is how we normally do arithmetic with numbers with many digits.
- Floating-point manages scaling automatically; the CPU keeps track of the radix point for you.

For many problems, floating-point arithmetic is by far the simplest approach.

31.4.1. Precision and Accuracy

Suggestion

You may want to review the discussion of precision and accuracy for fixed-point data in Section 29.10.1 on page 522.

Unlike the precision of packed decimal and binary integer values, floating-point values can be imprecise if the number of significant digits exceeds what the representation can provide. Because of the much greater range of floating-point values, they can sometimes retain greater accuracy.

Exercises

31.4.1.(1) In Figure 355 on page 559, represent Avogadro's Number and Planck's constant as numbers with one integer digit and several fraction digits, multiplied by a power of 10.

31.4.2.(2) Suppose you must represent both Avogadro's Number and Planck's constant in a fixed-point integer-fraction representation. Estimate the number of bits required.

31.5. Floating-Point Representations

A typical floating-point number representation has four elements: a sign, an exponent, its significant digits, and the *base* or *radix* used for the significant digits. Figure 356 shows a typical arrangement:

- one bit for the sign of the number (0 = +, 1 = −)
- some bits representing the exponent
- the remaining bits for the significant digits of the number, usually known as the *significand*.[196]

sign s	exponent e	significant digits (significand)

Figure 356. A typical floating-point representation

These fields need not be arranged in the order shown in Figure 356, but all System z floating-point representations use this basic format.

Designers of floating-point representations must then make several choices:

1. What base or radix should be used for the significant digits?
2. How should the significant digits be represented? Should they be digits of the radix (like the digits 0-9 for radix 10) or should they be encoded in some way?
3. Should a number like 12.34 be stored as the fraction 0.1234 with an exponent +2, or as the integer 1234 with an exponent −2, or with a single integer digit as in 1.234 with an exponent +1, or as 12.34 with a zero exponent?
4. Must all of the significant digits be present in the “significant digits” field, or can some of them be assumed, or combined with other fields?
5. If different data lengths are supported (such as 32 and 64 bits), should the same or different exponent widths be assigned to each?
6. How large a field should be allotted to the exponent? (A larger exponent field means less room for the significant digits.)

[196] The significand was sometimes called the “mantissa”, but that word had long been used for logarithms, where the meaning is quite different.

7. Should the exponent's value be represented in binary or some other radix? (The answer seems always to have been “binary”.)
8. How should the sign of the exponent be represented?

System z supports three sets of answers to these questions: hexadecimal, binary, and decimal,[197] which we'll investigate in Sections 33, 34, and 35.

To illustrate some possible choices, consider a floating-point representation of a number X, using 4 significant decimal digits. (We'll ignore what to do about the exponent for a moment.)

First, suppose the significant digits are stored as integers, with the radix point understood to follow the *rightmost* digit. In Figure 357, X is the value represented, and k is the exponent, a power of 10.

```
┌────────┐   ┌────────┐   ┌────────┐   ┌────────┐    ┌────────┐
│ 0 1 2 3.│   │ 2 3 4 0.│   │ 0 0 7 3.│   │ 0 0 7 3.│    │ 0 0 7 3.│
└────────┘   └────────┘   └────────┘   └────────┘    └────────┘
 k = 0        k = -1       k = 0        k = 2         k = -3
 X = 123      X = 234      X = 73       X = 7300      X = 0.073
```

Figure 357. An example of a floating-point representation using 4 decimal digits

Other digit strings and exponents can be used to represent the same value for X; for example, X=123 could be represented by the digits 1230 and exponent −1.

31.5.1. Left Normalization

Now, suppose the significant digits are stored as fractions, with the radix point understood to precede the *leftmost* digit. In Figure 358, X is again the value represented and k is the exponent as a power of 10.

```
┌────────┐   ┌────────┐   ┌────────┐   ┌────────┐    ┌────────┐
│.1 2 3 0 │   │.0 1 2 3 │   │.7 3 0 0 │   │.0 7 3 0 │    │.0 0 7 3 │
└────────┘   └────────┘   └────────┘   └────────┘    └────────┘
 k = 3        k = 4        k = 2        k = 3         k = 4
 X = 123      X = 123      X = 73       X = 73        X = 73
```

Figure 358. Another example of a floating-point representation using 4 decimal digits

The three representations of the number 73 show how the same number can have different representations.

Many floating-point representations viewing the significant digits as fractions expect the leftmost fraction digit to be nonzero: this means that as many *significant* digits as possible are kept with the number. We call this *left normalization*: the leftmost significand digit of a nonzero number is nonzero; we call the corresponding representation a “Floating-Point Fraction” or “FPF”. Thus, some values in Figures 358 and 359 are normalized and some are not.

```
Normalized    Unnormalized   Normalized    Unnormalized   Unnormalized
┌────────┐   ┌────────┐   ┌────────┐   ┌────────┐    ┌────────┐
│.1 2 3 0 │   │.0 1 2 3 │   │.7 3 0 0 │   │.0 7 3 0 │    │.0 0 7 3 │
└────────┘   └────────┘   └────────┘   └────────┘    └────────┘
 k = 3        k = 4        k = 2        k = 3         k = 4
 X = 123      X = 123      X = 73       X = 73        X = 73
```

Figure 359. A floating-point representation showing left normalized and unnormalized values

[197] Processor manufacturers previously created many different solutions (we'll see some in Section 36) but in recent years this wide variety of floating-point formats has converged on a small set of possibilities.

Figure 359 on page 561 shows that unnormalization allows redundant representations. We'll see that System z hexadecimal and decimal floating-point allow redundant representations, but binary floating-point does not.

31.5.2. Right Normalization

Now, suppose the significant digits are stored as integers, with the radix point understood to follow the *rightmost* digit. (We sometimes call this "right normalization".) In Figure 360, X is again the value represented and k is the exponent as a power of 10.

```
Unormalized    Normalized    Unnormalized   Unnormalized    Normalized
┌────────┐    ┌────────┐    ┌────────┐    ┌────────┐     ┌────────┐
│ 1 2 3 0.│    │ 0 1 2 3.│    │ 7 3 0 0.│    │ 0 7 3 0.│     │ 0 0 7 3.│
└────────┘    └────────┘    └────────┘    └────────┘     └────────┘
  k = -1        k = 0         k = -2        k = -1          k = 0
  X = 123       X = 123       X = 73        X = 73          X = 73
```

Figure 360. A floating-point representation showing right normalized and unnormalized values

The three representations of the number 73 show how the same number can have different representations. Right normalization was used on some early processors, but is not used for any System z floating-point representation.

31.5.3. No Normalization

Now, suppose the significant digits are viewed as integers, but with the radix point understood to to be "anywhere", as specified by the exponent. (For our convenience, we'll call this representation "Floating-Point NoNorm" or "FPN" for short.

In Figure 361, X is again the value represented and k is the exponent as a power of 10.

```
┌────────┐    ┌────────┐    ┌────────┐    ┌────────┐     ┌────────┐
│ 0 1 2 3 │    │ 2 3 4 0 │    │ 7 3 0 0 │    │ 0 7 3 0 │     │ 0 0 7 3 │
└────────┘    └────────┘    └────────┘    └────────┘     └────────┘
  k = 1         k = -1        k = -2        k = -1          k = 0
  X = 1230      X = 234       X = 73        X = 73          X = 73
```

Figure 361. A floating-point representation showing values without normalization

The three representations of the number 73 show how the same number can have different representations; in decimal floating-point terminology, any set of items with the same value but different representations is called a "cohort". We'll see how this works in Section 35.

31.5.4. Some Additional Details (*)

> **Interesting observations**
>
> This section will be interesting mainly to mathematically inclined readers.

Mathematically, floating-point numbers are a subset of the rational numbers: that is, they are a subset of all possible fractions (including fractions like 123/1). A value X is represented by $\pm M \times r^k$, where

M is an integer satisfying $0 \leq M < r^p$, an unsigned p-digit integer

r is the radix (or base) of the significant digits of the representation

p is the *precision* of M (the number of base-r digits)

k is the exponent, the power of r by which M is multiplied, typically positive *and* negative values of approximately equal ranges.

We sometimes say that the set of all such values is known as a floating-point system FP(r,p).

We can also use a *fraction* for the significant digits, so that $f = M \div r^p$, and the fraction satisfies $0 \leq f < 1.0$. This puts the radix point at the left end of the digits; X is then represented by $\pm f \times r^e$, and the exponent $e = k+p$ (k was the exponent we used for the "integer-format" representation above). The normalization condition then becomes $r^{-1} \leq f < 1.0$, so the most significant digit of f is nonzero.

System z uses both floating-point representations: hexadecimal and binary represent the significant digits as fractions, and decimal uses integers. To clarify which is being discussed, we'll use FPF(r,p) to mean a left-normalized fraction representation, FPI(r,p) to mean a right-normalized integer representation, and and FPN(r,p) to mean a non-mormalized integer representation. Thus, Figure 357 on page 561 and Figure 360 on page 562 use the FPI(10,4) representation, Figure 358 on page 561 uses the FPF(10,4) representation, and Figure 361 on page 562 uses the FPN(10,4) representation.

For example, suppose our FPF(10,4) representation allows a single decimal digit for the decimal exponent. The signed numbers +123 and +.0123 would be represented as in Figure 362:

Figure 362. Floating-point numbers with signed exponent

Another issue is rounding. If a number is not exactly representable in a given system, it should be approximated by one of the two nearest[198] representable values, preferably the value closest to the exact value. Figure 363 shows some familiar approximations in FPI(10,4):

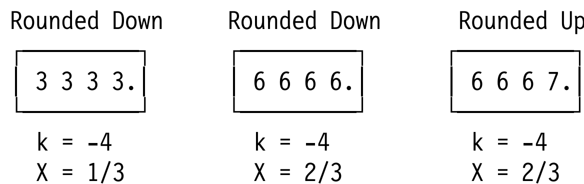


Figure 363. Examples of approximate floating-point representations

where we chose an approximation from one of the two nearest 4-digit neighbors of the exact value.

We saw in "31.2. Mixed Integer-Fraction Representation" on page 553 that decimal values like 0.1 cannot be converted exactly to binary or hexadecimal. So we must choose the best approximation of 0.1 in binary or hexadecimal floating-point. For example, if we use a hexadecimal floating-point representation with 6 significant digits, and a binary floating-point representation with 24 significant bits, the best resulting fraction values will be close, but not exact:

In FPF(16,6):	$1677722/16^6$	= 1677722/16777216	= 0.100 000 023 84
In FPF(2,24):	$13421773/2^{27}$	= 13421773/134217728	= 0.100 000 001 49

Most decimal fractions aren't representable precisely in base 2 or 16.

Exercises

31.5.1.(1) Using the FPF(10,4) representation in Figure 362 with a single signed decimal digit for the exponent, how many redundant representations of zero are possible?

31.5.2.(1) In the floating-point systems FPF(10,4), FPI(10,4), and FPN(10,4), how many representations are there of 4.7?

198 Some numbers are very difficult to convert accurately between bases. Modern conversions are almost always able to choose one of the two nearest neighbors of the exact value. This was not true for many years, and was the source of occasional unhappiness and confusion.

31.5.3.(2) In Figure 361 on page 562, sketch the position of the decimal point for each value of X.

31.6. System z Floating-Point Representations

System z supports three different representations for floating-point numbers: hexadecimal, binary, and decimal, with bases 16, 2, and 10.[199] For each representation, three data lengths are defined: 4 bytes, or *short*, 8 bytes, or *long*, and 16 bytes, or *extended*. Each has a sign bit, an exponent field, and a significand field for all of the significant digits (hexadecimal) or all but one of the significant digits (binary and decimal floating-point).

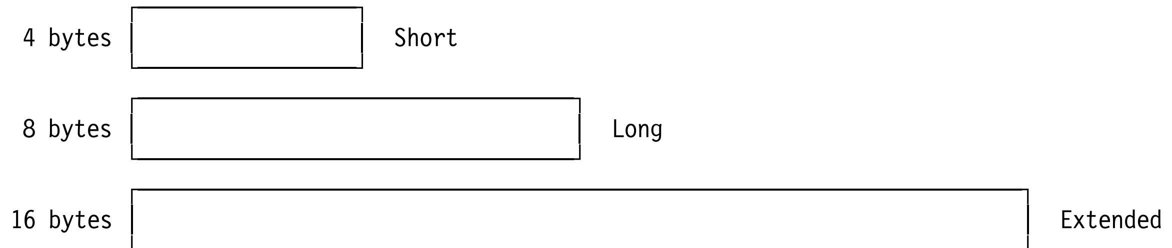


Figure 364. Three floating-point data lengths

The three System z floating-point number representations have different properties and ways of representing the data. We'll explore them in Sections 33, 34, and 35.

31.7. System z Floating-Point Registers

System z uses a separate set of registers for floating-point operands and arithmetic. For many years, the original System/360 and its successors supported only the four floating-point registers shown in Figure 365.

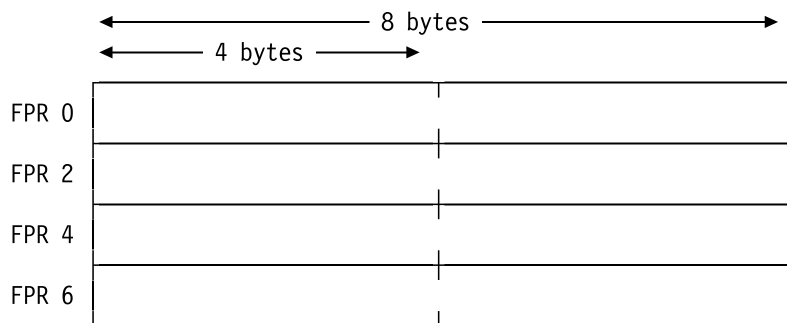


Figure 365. Four floating-point registers

This figure raises several questions:

1. Why were there only four floating-point registers while there are 16 general registers?
2. Why were the floating-point registers numbered 0, 2, 4, and 6 rather than 0, 1, 2, and 3?
3. Why are separate floating-point registers used for floating-point operations, rather than using the general registers?
4. The floating-point registers are 64 bits long; how can we access the rightmost 32 bits?

---

199 Historically, base 16 was supported first, then base 2, and then base 10.

Here are some possible answers:

1. In System/360 days, registers were expensive. Because businesses were the primary computer market, packed decimal data and operations were expected to satisfy most business needs. The “Floating-Point Feature” could be ordered if floating-point data and arithmetic was needed.
2. Because the floating-point registers are 64 bits long, their similarity to 32-bit general register pairs made it natural to number them the same way.
3. The general registers are needed for base registers and for address and integer arithmetic, so it seemed best to provide separate registers for floating-point operations.
4. Hexadecimal was the only floating-point representation supported for many years; short hexadecimal operands used just the leftmost 32 bits, while long operands used all 64 bits. There was rarely any need to access only the rightmost 32 bits of the floating-point register.

Later, IBM added the remaining 12 floating-point registers; all modern CPUs have 16.

Unlike the general registers, the floating-point registers are addressed *only* by their leftmost portions. The low-order portion of the register is not separately addressable.

In Figure 364 on page 564, extended-precision operands are shown as being 16 bytes long; because the floating-point registers are only 8 bytes long, a *pair* of registers is used.[200] The eight valid register pairs are (0,2), (1,3), (4,6), (5,7), (8,10), (9,11), (12,14), and (13,15), as illustrated by the “bracketing” in Figure 366 on page 566.

Remember!

Each 64-bit floating-point register is *not* a pair of 32-bit registers.

[200] We've already seen general register pairing for double-length integer products and dividends.

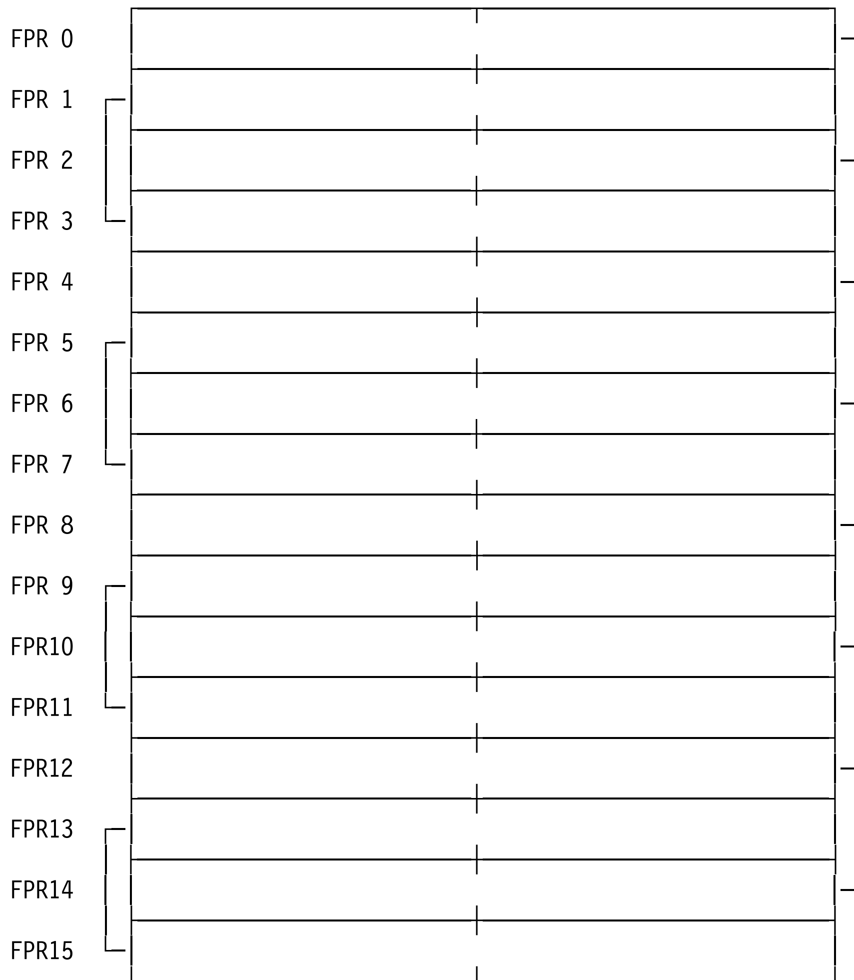


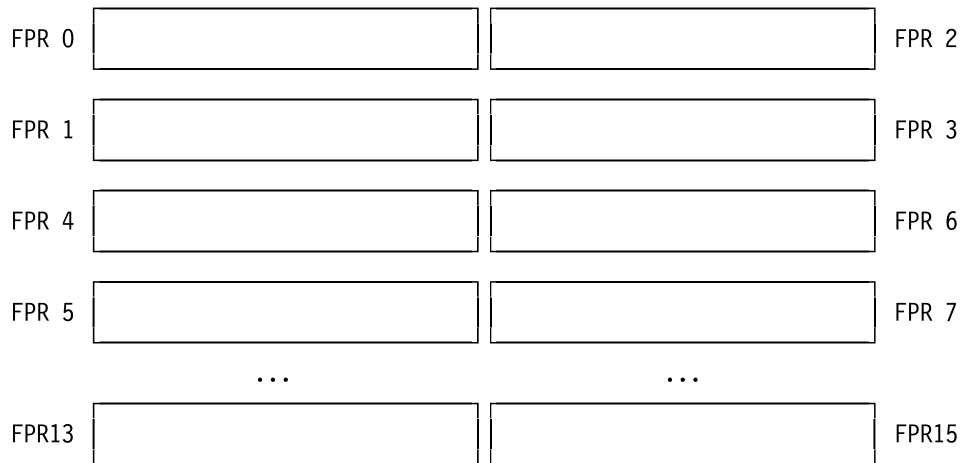
Figure 366. All sixteen floating-point registers, showing register pairings

The high-order half of an extended operand is in the lower-numbered register, and the low-order half is in the register number two higher.

Remember!

floating-point register pairs are R_n and R_n+2, not R_n and R_n+1, as with general register pairs.

It may help to visualize floating-point register pairing this way:



You may want to be careful when referencing floating-point registers on a very old CPU that has only the four original registers. A specification exception would be generated by any instruction which tries to refer to any but FPR0, FPR2, FPR4, and FPR6.[201]

Exercises

31.7.1.(1) Why do you think the floating-point registers are not paired in the same way as the general registers? For example, (0,1), (2,3), etc.

31.7.2.(2) Suppose the CPU has the number of a floating-point register to be used in an extended precision operation. What is an easy way to derive the number of the paired floating-point register?

31.8. Floating-Point Constants

The Assembler's DC statement lets you define floating-point constants of the three standard lengths, with these default alignments:

Type	Length
E	4 bytes, fullword aligned
D	8 bytes, doubleword aligned
L	16 bytes, doubleword aligned

and all three representations, hexadecimal, binary, and decimal, specified by these subtypes:

Subtype	Data Representation
H or none	Hexadecimal floating-point
B	Binary floating-point
D	Decimal floating-point

These are described in Sections 33, 34, and 35, discussing each type of representation and arithmetic.

201 The Assembler can help by checking register references at assembly time. By default, all 16 registers are valid, but if you want to be sure your program references only those four, see the “AFPR” topic in the Assembler's *Language Reference* manual.

Exercises

31.8.1.(1) What DC constant type (and subtype, if any) would you specify for floating-point constants of these types?

- Long decimal
- Short binary
- Extended hexadecimal
- Extended decimal

31.9. Representation-Independent Floating-Point Instructions

While many of the floating-point instructions depend on the representation of the data they manipulate, some work for all three.[202] That is, you can put any string of bits in a floating-point register; just be careful not to do arithmetic with them!

Worth remembering

These instructions do not depend on the representation of the data in the floating-point registers.

31.9.1. Register-Storage Instructions

The instructions in Table 221 move data between memory and the floating-point registers for data in all three System z floating-point representations. They do the same operations for floating-point register data as do the related instructions (like L, ST) for general register data. The CC is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
78	LE	RX	Load (Short)	ED64	LEY	RXY	Load (Short)
68	LD	RX	Load (Long)	ED65	LDY	RXY	Load (Long)
70	STE	RX	Store (Short)	ED66	STEY	RXY	Store (Short)
60	STD	RX	Store (Long)	ED67	STDY	RXY	Store (Long)

Table 221. Basic floating-point instructions

There are no register-memory instructions to load or store an extended operand into or from a floating-point register pair, and no instructions that load or store multiple floating-point registers.

31.9.2. Register-Register Instructions

Table 222 lists the instructions for moving floating-point operands among the floating-point registers:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
38	LER	RR	Load FPR from FPR (Short)	28	LDR	RR	Load FPR from FPR (Long)
B365	LXR	RRE	Load FPR from FPR (Extended)				
B373	LCDFR	RRE	Load Complement (Long)	B371	LNDFR	RRE	Load Negative (Long)
B370	LPDFR	RRE	Load Positive (Long)				

Table 222. Instructions copying data between FPRs

[202] The *z/Architecture Principles of Operation* refers to them as “radix-independent” or “Floating-Point Support” instructions.

You may have noticed that “Load and Test” doesn't appear among these representation-independent instructions; we'll see why such instructions depend on the format of the operands.

LER and LDR are similar to LR and LGR: both LR and LER move only 4 bytes, and LGR and LDR move 8 bytes. LXR moves all 16 bytes, and requires both operands to refer to the lower-numbered register of a floating-point register pair.[203]

There are two important differences between LR and LER:

1. While LR copies the *low-order* 4 bytes of one general register to another, LER copies the *high-order* 4 bytes of one floating-point register to another. For example:

```
LG    0,=X'0123456789ABCDEF'  Load GG0
LG    2,=X'FEDCBA9876543210'  Load GG2
LR    0,2                     c(GG0)=X'0123456776543210'

LD    0,=X'0123456789ABCDEF'  Load FPR0
LD    2,=X'FEDCBA9876543210'  Load FPR2
LER   0,2                     c(FPR0)=X'FEDCBA9889ABCDEF'
```

   The *rightmost* 4 bytes of GG0 are changed, but the *leftmost* 4 bytes of FPR0 are changed.

2. There is no way to reference the 4 low-order bytes of a floating-point register separately from its high-order 4 bytes.

The LCDFR, LNDFR, and LPDFR instructions in Table 222 on page 568 are different from many similar instructions (such as LCR, LNR, and LPR for binary integers, and LCER, LNER, and LPER for hexadecimal floating-point), because they *don't* set the Condition Code. There are other representation-dependent instructions like these that *do* set the CC.

31.9.3. Load-Zero Instructions

Before these instructions were introduced, setting a floating-point register to zero was awkward: either a zero had to be loaded from memory (which is relatively slow), or the register's content had to be subtracted from itself, which could also be slow (and sometimes cause other undesired side-effects). The three instructions in Table 223 simply set the designated operand register(s) to zero, and do not change the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B374	LZER	RRE	Load Zero (Short)	B375	LZDR	RRE	Load Zero (Long)
B376	LZXR	RRE	Load Zero (Extended)				

Table 223. Floating-point Load Zero instructions

The LZER instruction probably isn't needed, because LZDR will clear the full 8 bytes of the floating-point register; and because you can't refer separately to the low-order 4 bytes of the register, there seems to be little gain in clearing only the high-order 4 bytes.

31.9.4. GPR-FPR Copying Instructions

Transferring data between general registers and floating-point registers on System/360 required using an intermediate storage area in memory. As CPUs have become much faster relative to memory speeds, those two memory accesses can cause program delays.

The two instructions in Table 224 on page 570 transfer data directly between general and floating-point registers.

[203] Previously, two LDR instructions were needed to copy an extended-precision operand from one floating-point register pair to another.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3C1	LDGR	RXY	Load FPR from GPR (Long)	B3CD	LGDR	RXY	Load GPR from FPR (Long)

Table 224. Instructions moving data between FPRs and GPRs

As the instruction names indicate, these two instructions move 64-bit operands. For example:

```
         LDGR  4,13             Copy c(GG13) to c(FPR4)
         LGDR  1,6              Copy c(FPR6) to c(GG1)
```

31.9.5. Sign-Copying Instruction

The Copy Sign instruction is unusual: the second operand is copied to the first operand (as for LDR), but with the sign bit of the third operand!

Op	Mnem	Type	Instruction
B372	CPSDR	RRF	Copy Sign (Long)

Table 225. Copy Sign instruction

The instruction format is

$CPSDR\ R_1,R_3,R_2$

The second operand (in R_2) is copied to the first operand (in R_1) with the sign of the third operand (in R_3). Only the long format is supported; but you can use CPSDR for short operands, and for extended operands if you remember to copy the low-order half from the second operand's higher-numbered register to the higher-numbered register of the first operand.

Thus, you could write

```
         CPSDR 1,5,8            Copy FPR8 to FPR1 with FPR5's sign
```

to avoid having to write an LDR instruction followed by other Load and Test and Load Complement instructions to set the correct sign.

Exercises

31.9.1.(1) Write three different short instruction sequences to swap the contents of FPR0 and FPR4.

31.9.2.(1) In Exercise 19.5.1, you showed how the contents of two general registers could be exchanged using logical operations. Show another way to do this without referencing memory or by using logical operations.

31.9.3.(1) If you designed register-memory instructions to load and store extended floating-point operands, or to load and store multiple floating-point registers, what mnemonics might you assign to those four instructions? What possible exception conditions might they recognize?

31.10. Summary

This section has described scaled fixed-point arithmetic, and shown why its difficulties led to the introduction of floating-point.

The representation-independent instructions useful in programs handling floating-point operands are listed in Table 226 on page 571.

Operation	Operands		
	4 bytes	8 bytes	16 bytes
Load (memory)	LE LEY	LD LDY	
Load (register)	LER	LDR LCDFR LNDFR LPDFR	LXR
Zero (register)	LZER	LZDR	LZXR
Copy Sign (register)		CPSDR	
Store (memory)	STE STEY	STD STDY	

Table 226. Basic Load/Store instructions for floating-point operands

The two instructions in Table 227 transfer data between floating-point registers and general registers.

Instruction	Operand 1	Operand 2
LGDR	8-byte GPR	8-byte FPR
LDGR	8-byte FPR	8-byte GPR

Table 227. Instructions moving operands between GPRs and FPRs

---

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
CPSDR	B372
LCDFR	B373
LD	68
LDGR	B3C1
LDR	28
LDY	ED65
LE	78

Mnemonic	Opcode
LER	38
LEY	ED64
LGDR	B3CD
LNDFR	B371
LPDFR	B370
LXR	B365
LZER	B375

Mnemonic	Opcode
LZDR	B375
LZXR	B376
STD	60
STDY	ED67
STE	70
STEY	ED66

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
28	LDR
38	LER
60	STD
68	LD
70	STE
78	LE
B365	LXR

Opcode	Mnemonic
B370	LPDFR
B371	LNDFR
B372	CPSDR
B373	LCDFR
B374	LZER
B375	LZDR
B376	LZDR

Opcode	Mnemonic
B3C1	LDGR
B3CD	LGDR
ED64	LEY
ED65	LDY
ED66	STEY
ED67	STDY

Terms and Definitions

exponent
The power of the radix by which the significand of a floating-point number must be multiplied to determine its value.

floating-point
A data representation with a sign, an exponent, and a set of significant digits.

floating-point system FP(r,p)
A floating-point data representation with a specified radix and number of significant digits, denoted FPF(r,p) or FPI(r,p).

floating-point system FPF(r,p)
A floating-point system with radix r and p digits of precision, in which the significant digits are represented as fractions.

floating-point system FPI(r,p)
A floating-point system with radix r and p digits of precision, in which the significant digits are represented as integers.

mantissa
An old term for the significant digits of a floating-point number.[204]

radix
The base in which the significant digits of a floating-point number are represented.

significand
The numerically significant digits of a floating-point number, whether explicitly or implicitly represented.

Programming Problems

Problem 31.1.(1) Write a program using packed decimal arithmetic to solve the import-tax calculation described in Section 31.3, and print the formatted result.

Problem 31.2.(2) Write a program using fixed binary arithmetic to solve the import-tax calculation described in Section 31.3, and print the formatted result.

Problem 31.3.(4) Write a program using fixed binary arithmetic to evaluate and print the square root of 2 using the Newton-Raphson iteration described in Section 31.3.1, with initial estimate 1. (The hardest part of this problem will probably be formatting the fraction part of the result.)

---

[204] This term should be avoided because it can be confused with the mantissa of a logarithm, which is quite different.

32. Basic Concepts of Floating-Point Arithmetic

```
 3333333333    2222222222
333333333333  222222222222
33        33  22        22
          33            22
          33            22
       3333            22
       3333          22
          33       22
          33     22
33        33   22
333333333333  222222222222
 3333333333   222222222222
```

In this section we examine the basic behavior of floating-point arithmetic, and some factors that you (and CPU designers) must consider. For the following examples, we'll use a floating-point format where the significant digits are decimal fractions: the radix point is at the left end of the most significant digit. We'll start with floating-point multiplication.

32.1. Floating-Point Multiplication

Suppose we want to multiply two floating-point numbers A and B, where $A = a \times 10^{Ea}$ and $B = b \times 10^{Eb}$. Their product is

$$A \times B = (a \times b) \times 10^{(Ea+Eb)}$$

That is, we multiply the significant digits and add the exponents.

For example, suppose A is 3300 (33×10^{2}) and B is 0.029 (29×10^{-3}). Multiplying, we find that

$$A \times B = (33 \times 29) \times 10^{(2+(-3))} = 957 \times 10^{-1} = 95.7$$

If we represent A and B in FPF(10,4) as four-digit base-10 floating-point values with normalized fractions, $A = 0.3300 \times 10^{4}$ and $B = 0.2900 \times 10^{-1}$. Multiplying gives

$$A \times B = (0.3300 \times 0.2900) \times 10^{(4+(-1))} = 0.0957 \times 10^{+3} = 95.7$$

The result is numerically correct, but the product fraction 0.0957 is not normalized. To generate a normalized result, the fraction must be shifted left one digit position to form 0.9570; and because this increases the fraction by 10, we must reduce the exponent by 1 to compensate. The normalized result is then 0.9570×10^{2}.

This final multiplication correction is called *post-normalization*, and may be needed if the leading digit of the product fraction is zero. For example, if we multiply the decimal fractions 0.2 and 0.3, the product fraction 0.06 must be shifted left once to obtain a normalized result. If the post-normalizing shift is performed, the exponent is reduced by 1 to account for the shift having multiplied the fraction by 10.

This example shows that multiplication using FPF(r,p) may require a post-normalization shift. The same product using integer values in FPI(10,4) would give

$$A \times B = (0033. \times 10^{2}) \times (0029. \times 10^{-3}) = 0957. \times 10^{2+(-3)} = 0957. \times 10^{-1} = 95.7$$

and no post-normalizing shift is needed.

The products in these examples have had few enough digits that we haven't had to worry about excess low-order (or high-order) digits. We'll investigate what can be done about them in Section 32.3.

Exercises

32.1.1.(2) In FPF(10,4) using 4-digit internal arithmetic, what are the normalized values of these products?

1. $(0.7654\times10^{12})\times(0.1235\times10^{-8})$
2. $(0.7655\times10^{2})\times(0.7655\times10^{2})$
3. $(0.0333\times10^{2})\times(0.0321\times10^{1})$

32.2. Pre-Normalization of Fraction Operands

Our examples assumed that the operands of a product in an FPF(10,4) representation were normalized, so that at most a single post-normalizing shift is needed. But what if the operands are *not* normalized? If the numbers A and B had been represented by the unnormalized values $A = 0.0033\times10^{6}$ and $B = 0.0029\times10^{1}$, the product will be

$$A\times B = (0.0033\times0.0029)\times10^{6+1} = 0.00000957\times10^{7}$$

If we had retained only the first 4 digits of this product, all significance would have been lost. This product fraction has *five* leading zeros, so something must be done to preserve the accuracy of the result. This is called *pre-normalization*: before the operands are multiplied the CPU internally normalizes them and adjusts their exponents; the product fraction is then 0.0957, which needs only a single post-normalizing shift.

Because most floating-point operands are already normalized, the cost of pre-normalization is very low relative to the other steps needed to complete the instruction. Also, if the operand fractions are pre-normalized, we know that at most one post-normalization shift may be needed.

Exercises

32.2.1.(2)+ In FPF(r,p), what is the largest number of pre-normalizing shifts required for a nonzero operand?

32.2.2.(2) Show the normalized results in FPF(10,4) of each product, with and without pre-normalized operands.

1. $(0.0147\times10^{2})\times(0.0070\times10^{3})$
2. $(0.0073\times10^{4})\times(0.0071\times10^{1})$

32.3. Floating-Point Rounding

We learned to round decimal results by adding 5 to the first digit to be "lost", propagating any carries. This is sometimes called "arithmetic rounding" or "round half-up" (among other names). It is one of many ways to round a more precise value to a less.

Arithmetic rounding is "biased" in the sense the sums may be slightly too large. For example, suppose you are calculating the sum of the exact decimal values .11115, .11125, .11135, .11145, and .11155, and each exact value must be rounded to four significant digits before final summation. With arithmetic rounding, the intermediate values are .1112, .1113, .1114, .1115, and .1116 and their sum is .5570. But the sum of the 5-digit intermediate values is .55675, which we would round to 4 significant digits as .5568.

In hexadecimal floating-point, arithmetic results are truncated (“rounded toward zero”). Then, our five truncated intermediate decimal values would be .1111, .1112, .1113, .1114, and .1115, and their sum would be .5565.

An important rounding that reduces bias is “round half-even” (sometimes called “banker's rounding”). If the value to be rounded lies exactly half way between two closest neighboring numbers in the less-precise representation, we choose the value whose low-order digit is even. With half-even rounding, our five intermediate values would be .1112, .1112, .1114, .1114, and .1116, and their sum is .5568; and this is also the rounded (half-even or arithmetic rounding) value of the exact sum.

Starting with the exact value, we can summarize our examples of these three types of rounding:

- .55675 = exact
- .5565 = truncated (rounded toward zero)
- .5570 = arithmetic (half-up) rounding
- .5568 = half-even rounding

This shows how arithmetic results can depend on the type of rounding used. System z supports some other types of rounding, especially for binary and decimal floating-point arithmetic. We noted above that hexadecimal floating-point arithmetic truncates: its sum would be .5565.

As we'll see in Section 32.4, the guard and rounding digits are used only internally, and are *not* a part of any accessible register. This means that these two digits are not available on completion of an operation, and their values are not carried over from one instruction to the next. You can think of them as being set to zero at the beginning of the operation.

Exercises

32.3.1.(2)+ Given the two FPF(10,4) numbers $X=.1000\times10^1$ (=1.000) and $Y=.5555\times10^0$ (=.5555), show the results of four iterations of the calculation

```
X = (X+Y)-Y
```

using (1) round to zero, (2) round half-up, and (3) round half-even arithmetic for each iteration.

32.4. Guard and Rounding Digits (*)

Let's try other examples of multiplication in FPF(10,4) with normalized operands. First, consider $1\times1 = (.1000\times10^1)\times(.1000\times10^1)$. The intermediate product is $.0100\times10^2$, which after normalization becomes $.1000\times10^1$.

Now, consider $1\times1.234 = (.1000\times10^1)\times(.1234\times10^1)$. Since our products are 4 digits long, the intermediate product is $.0123\times10^2$, or $.1230\times10^1$ after normalization. But this is unacceptable: we want to believe that 1×A for any A always yields A.

The solution to this unfortunate situation[205] is to use a *guard digit*, a single extra digit used internally for intermediate results. With a guard digit, the intermediate product is .0123[4], where the guard digit is shown in square brackets [ ]. The post-normalizing shift shifts the intermediate product *including* the guard digit left by 1 digit position, and the final product is $.1234\times10^1$ as required.

Another example can reinforce the importance of a guard digit. If we have two positive numbers A and B with $A < B$, then multiplying both by A should (according to the rules of algebra) mean that $A^2 < AB$. Now, suppose $A = .9999\times10^0$ and $B = .1000\times10^1$, so $A < B$. Without a guard digit, the products are $A^2 = .9998\times10^0$, and $AB = .9990\times10^0$, which means that

205 It happened to IBM in 1966, because the very first System/360 processors omitted the guard digit for long hexadecimal arithmetic. The oversight was quickly corrected, along with some other hexadecimal floating-point inconsistencies.

A^2 > AB instead! *With* a guard digit, the products are A^2 = $.9998\times10^0$ and AB = $.9999\times10^0$, preserving the required inequality.

A guard digit is needed not only to improve the precision of a result. It might seem that the difference between the results with and without a guard digit is small (and especially so with fractions having many digits), and is therefore negligible. As we saw in the previous two examples, the important observation is *not* that the answer is slightly incorrect, but that an expected *ordering* relationship between the operands might be destroyed. It's quite difficult to write reliable programs if you can't assume that inequalities between operands behave normally.

Suppose we multiply 509×151; the product is 76859. In FPF(10,4), $(.5090\times10^3)\times(.1510\times10^3)$ gives an intermediate product $.0768[5]9\times10^6$, and post-normalization gives the result $.7685\times10^5$ = 76850. But a much better approximation in FPF(10,4) is $.7686\times10^5$ = 76860. To produce this more accurate result, a second internal digit is needed, a *rounding digit*.

To round, we first post-normalize the intermediate result and then add 5 to the digit (if any) after the last significant digit (propagating carries, of course). The rounded intermediate product fraction .0768[5]9 becomes

```
 .76859  (normalized)
+     5
 .76864
```

The rounding digit is then discarded, giving $.7686\times10^5$, as desired.

If no post-normalizing shift is needed, the guard digit is used as the rounding digit.

In System z processors, hexadecimal floating-point arithmetic provides a guard digit but no rounding digit. Special instructions can round longer results (extended and long) to shorter (long and short), but very few hexadecimal floating-point instructions round their results.

However, System z binary and decimal floating-point arithmetic instructions let you specify several *rounding modes*, both generally and for individual instructions. We'll explore the instructions and rounding modes when we discuss binary and decimal floating-point.

Exercises

32.4.1.(2) Without a guard digit, what types of operand A in FPF(10,4) will cause 1×A to be unequal to A?

32.4.2.(2) For these pairs of FPF(10,4) operands, show their products (a) with no guard or rounding digit, (b) with a guard digit but no rounding digit, and (c) with both guard and rounding digits.

1. 155×165 (=25575)
2. 45×2469 (=111105)
3. 21×1117 (=23457)
4. 127×137 (=17399)

32.4.3.(3) For these pairs of FPF(10,4) operands, show their products (a) with no guard or rounding digit, (b) with a guard digit but no rounding digit, and (c) with both guard and rounding digits.

1. 509×101 (=51409)
2. 509×555 (=282495)
3. 509×550 (=279950)
4. 407×515 (=209605)

32.4.4.(2) Repeat Exercise 31.1.1, assuming the operands are pre-normalized. Show the results (a) with a guard digit, and (b) with guard and rounding digits.

32.4.5.(3) Construct three examples of products using FPF(10,4) operands that show differences between rounding *before* post-normalization and rounding *after* post-normalization.

32.4.6.(2)+ Suppose you must multiply these two floating-point numbers (represented as three decimal fraction digits): 0.124 and 0.456. You must then test whether their product is greater than 0.0562. Show the results of the comparison if the multiplication is done (a) without a guard digit, and (b) with a guard digit.

32.5. Integer-Based Representations (*)

Integer-based floating-point representations raise a different set of questions. Using the FPI(10,4) notation of Figure 357 on page 561, we can consider several representations of the number 73:

```
┌────────┐    ┌────────┐    ┌────────┐
│7 3 0 0.│    │0 7 3 0.│    │0 0 7 3.│
└────────┘    └────────┘    └────────┘
 k = -2        k = -1        k = 0
 X = 73        X = 73        X = 73
```

Figure 367. Integer-based representation of 73 in FPI(10,4)

All three representations have the same value, but only the third "looks like" the integer 73, because its exponent is zero (so the lowest-order digit is multiplied by 10^0).

Now, suppose we multiply 73 by 730, both represented as integers with exponent zero. then $(0073.\times10^0)\times(0730.\times10^0)$ is $53290.\times10^0$. But there are five digits in the product, so to preserve precision we must shift the result one digit to the right, giving $5329.\times10^1$. The result no longer has the original integer representation, because the lowest-order digit has effectively been multiplied by 10^1. Because the discarded digit was zero, we didn't worry about rounding. Thus, the product is

```
┌───────┐
│5 3 2 9│ 0.
└───────┘
 k = +1
 X = 53290
```

and the decimal point lies beyond the precision of the representation.

Now, consider the product $127\times137 = 17399$. If we start with integer values with exponent zero, the product again has too many digits and must be shifted right once to fit in the 4-digit significand field. Because the digit to be discarded is nonzero, the remaining digits should be rounded, so the result will be $1740.\times10^1$.

Integer-based floating-point representations are not "normalized" either by requiring the highest-order digit to be nonzero (as in fraction-based representations), nor by requiring the lowest-order digit to be nonzero. Integer-based floating-point is used in System z only for decimal floating-point, which we'll explore in Section 35.

Exercises

32.5.1.(2) In FPI(r,p), what advantage might there be if you pre-normalize the nonzero operands of a multiplication "to the right" so that the lowest-order digit is nonzero?

32.6. Floating-Point Division

If we calculate the quotient Q of two floating-point operands S and T, $Q = S \div T$, where $S = s \times 10^{Es}$ and $T = t \times 10^{Et}$, the quotient $S \div T$ will be

$$S \div T = (s \div t) \times 10^{(Es-Et)}$$

To illustrate, suppose we use the same FPF(10,4) representation for S and T as for A and B above (in Section 32.4): four fractional base-10 digits and an exponent, and let S = 3300 (0.3300×10^4) and T = 6 (0.6000×10^1), so the quotient will be 550. Then

$$S \div T = (0.3300 \div 0.6000) \times 10^{(4-1)} = 0.5500 \times 10^3 = 550$$

In this case, the quotient fraction is properly normalized.

Now, suppose we divide 0.5 by 0.2, ignoring exponents. The result “fraction” is 2.5, which is greater than 1, so the result requires a corrective right shift.[206] Because shifting the fraction right by one digit position is the same as dividing it by 10, the exponent of the result is increased by 1 to compensate for the shift. This is illustrated in Figure 368:

```
┌─────────┐     ┌─────────┐      ┌─────────┐     ┌─────────┐
│.5 0 0 0 │  ÷  │.2 0 0 0 │  = 2 │.5 0 0 0 │ ──► │.2 5 0 0 │
└─────────┘     └─────────┘      └─────────┘     └─────────┘
   k = 0           k = 0            k = 0           k = 1
```

Figure 368. Illustrating floating-point division corrective right shift

Division operand fractions are pre-normalized, as for multiplication. If they weren't, we might divide 0.95 by 0.0005, giving 1900. for the (very unnormalized) quotient! This could cause extra work to produce a correctly normalized result; possibly, the CPU designers would avoid designing the extra circuits and simply indicate an error.

If the divisor T is zero, the division is improper and the CPU generates a floating-point divide exception. If the dividend S is zero, the result can be set to zero immediately.

Exercises

32.6.1.(2) Ignoring exponents, which of the following division operations will require a corrective right shift for the 4-digit quotient? Show the rounded 4-digit quotient fraction.

1. .1428 ÷ .7142
2. .6667 ÷ .6666
3. .1277 ÷ .3456
4. .3456 ÷ .1275
5. .9999 ÷ .9999

32.7. Floating-Point Addition and Subtraction

Adding and subtracting floating-point numbers is more complicated than multiplying and dividing. We don't need to consider subtraction separately: to subtract, the CPU needs only to invert the sign bit of the second operand and then add. Thus, we consider adding operands with like signs or unlike signs.[207]

One major difference between division and multiplication on the one hand, and addition on the other, is that in addition neither operand needs to be pre-normalized. This is because

[206] This corrective right shift is not a post-normalizing shift in same sense as for multiplication, because “normalization” is usually understood to imply a *left* shift of the fraction to eliminate leading zero digits.

[207] The CPU treats addition of operands with unlike signs as just another name for subtraction.

- there is no guarantee (as was the case for multiplication and division) that at most one corrective shift will be needed when the addition is completed, and
- some calculations are simplified by *not* pre-normalizing; some of these will be illustrated when we discuss hexadecimal floating-point in Section 33.

To illustrate floating-point addition, we'll again use decimal operands to add 124 and 3. In FPF(10,4), these are represented as 0.1240×10^3 and 0.3000×10^1 respectively. Now we can't simply add the fractions directly, since the true positions of the decimal points don't correspond to the same leading digit positions: that is, adding

```
0.1240 + 0.3000 =(?) 0.4240
```

is clearly not correct. To add the fractions correctly, we must compare the exponents of the operands: if they are equal, the fractions can be added immediately. If the exponents are not equal, the fraction part of the number with the *smaller* exponent is shifted right by a number of digit positions equal to the difference between the exponents. Because each right shift means that the smaller exponent is increased by 1, it will eventually be equal to the larger.

Thus, the CPU must *unnormalize* the fraction of the operand with the smaller exponent until the exponents of the two operands are equal. The fraction parts can then be added directly; in our example we would find after shifting the fraction 0.3 to the right by $3-1 = 2$ places and adding, that the result fraction is

Before	After
$.1240 \times 10^3$	$.1240 \times 10^3$
$+\ \underline{.3000} \times 10^1$	$+\ \underline{.0030} \times 10^3$
$????? \times 10^?$	$.1270 \times 10^3$

The result exponent is 3, the exponent of the larger operand, and the value to which both exponents were adjusted.

As another decimal example of adding operands with like signs, suppose we want to add 62 and 77, which would be represented as $.6200 \times 10^2$ and $.7700 \times 10^2$. The exponents are equal, so the fraction parts may be added directly, giving 1.3900×10^2.

$$
\begin{array}{r}
.6200 \times 10^2 \\
+\ \underline{.7700} \times 10^2 \\
1.3900 \times 10^2\text{, or } +.1390 \times 10^3.
\end{array}
$$

This "fraction" is not less than one in magnitude, so a single corrective right shift is needed; the exponent is increased by one to account for the right shift. Thus, the result is $+.1390 \times 10^3$, as desired.

When adding two normalized numbers of *like* signs, the only post-addition correction that may be needed is a possible right shift of the fraction by one digit position, with an accompanying increase of the exponent or characteristic by one.

What happens if the operands have exponents with a larger difference? Suppose we add .1234 and .00008, which in FPF(10,4) have representations $.1234 \times 10^0$ and $.8000 \times 10^{-4}$. Because the exponent difference is 4, we must shift the smaller operand right 4 places:

$$
\begin{array}{r}
.1234\ \ \times 10^0 \\
+\ \underline{.00008} \times 10^0 \\
.12348 \times 10^0
\end{array}
$$

As with multiplication, discarding the extra rightmost digit would give a less accurate result; some types of floating-point arithmetic use the extra digit position to round the result to $.1235 \times 10^0$.

If the exponent difference is large enough, the smaller operand would be shifted so far to the right that its digits can't contribute to the sum. Thus, the CPU can compare exponents to quickly determine that it can deliver the larger operand as the result without doing any arithmetic.

Adding numbers with unlike signs proceeds the same way: the fraction part of the operand with the smaller exponent is right-shifted a number of digit positions equal to the difference of the exponents. The subtraction ("negative addition") of the two fractions is then performed, and the sign of the result is noted.

Since the result fraction can be smaller in magnitude than the larger of the two original operands, we may find some leading zero digits in the result. If a normalized result is desired, the fraction digits are shifted left (*post-normalized*) and the exponent is decremented by the number of left shifts performed. (Sometimes it's useful to retain the unnormalized result; we'll see examples when we discuss hexadecimal floating-point.)

For example, suppose we add +72 ($+.7200\times10^2$) and −67 ($-.6700\times10^2$). Because the exponents are equal, no operand shift is needed. Then, the steps of the addition are these:

```
+ .7200 × 10^2
- .6700 × 10^2
+ .0500 × 10^2
```

This example shows that adding numbers of unlike signs can generate an unnormalized result. If a normalized result is required, the result would be $+.5000\times10^1$, as expected. Similarly, we can subtract 99 ($.9900\times10^2$) from 102 ($.1020\times10^3$). After shifting the operand with the smaller exponent to the right by one digit position, the steps are shown in this example:

```
+ .1020 × 10^3
- .0990 × 10^3
+ .0030 × 10^3, or .3000 × 10^1.
```

and two shifts are needed to normalize the result.

Exercises

32.7.1.(2) For each of these differences in FPF(10,4) arithmetic, show the result (a) without guard and rounding digits, (b) with a guard digit but no rounding digit, and (c) with both rounding and guard digits.

1. $(.7435\times10^3)-(.9621\times10^1)$
2. $(.7435\times10^1)-(.6994\times10^1)$
3. $(.1043\times10^5)-(.9527\times10^4)$
4. $(.1000\times10^0)-(.9992\times10^{-2})$

32.7.2.(1) In describing addition with unlike signs, the text above states "... the result fraction *can be* smaller in magnitude than the larger of the two original operands...". Why does it not state "... the result fraction *is* smaller in magnitude..."?

32.7.3.(2)+ In FPF(10,4), show the intermediate steps and the results in calculating the following quantities. (1) 9*9, (2) 9−9, (3) 643−552, (4) 2/3, (5) 28−32.

32.8. Floating-Point Precision

You may have learned in a mathematics class about what mathematicians call "real" numbers that have unlimited precision[208] and magnitude. But in our "realistic" world, we must deal with numbers having finite precision — limited numbers of digits. This is familiar to us; the import-tax example in Section 31.3 on page 557 is typical of everyday calculations.

Floating-point arithmetic is necessarily "realistic" because data items have finite numbers of digits. When we want to know the precision of a floating-point number, we need to know the precision p of its representation.

We'll use values in FPF(10,4) to illustrate. Ignoring exponents for the moment, consider the fraction `.1000`: we might say it has a precision of 4 decimal digits. The next larger value is `.1001`; the *relative* difference between the two is `.0001 ÷ .1000`, or 10^{-3}. Somehow, our four digits of precision has become three!

[208] If you ever had to follow a mathematics proof using "deltas" and "epsilons", you'll remember that they can be arbitrarily tiny.

The relative size of a one-digit change in the rightmost digit of a floating-point number is sometimes called a “**U**nit in the **L**ast **P**lace”, or an “ulp” for short. An ulp is defined by

```
ulp(x) = successor(x) - x
```

where the successor of a floating-point number is the next larger value.[209] This is the “spacing” between neighboring values.

Because the size of an ulp depends on the size of the number, it may be more useful to know its relative size. The *relative* size of an ulp is

```
ulp(x) ÷ x
```

This is the relative weight of the low-order digit of the number x.

In FPF(10,4), ulp($.1000\times10^{6}$) is 10^{2}, and ulp($.1000\times10^{-6}$) is 10^{-10}; but the relative size of both ulps is about 10^{-3}.

But now consider the fraction .9998: its successor (the next larger value) is .9999, so their relative difference is .0001 ÷ .9998, or almost 10^{-4}. So, the relative size of an ulp can vary by as much as a factor of 10, the radix of the representation.

This behavior applies to any floating-point representation using radix r: the relative size of an ulp can vary between r^{-p} and $r^{-(p-1)}$, so its size can vary by as much as r.[210] Thus the relative precision of a floating-point representation is best described as $r^{-(p-1)}$, and the precision of a p-digit decimal fraction is *not* 10^{-p}, but $10^{-(p-1)}$.

As we saw for values in FPF(10,4), the precision of a floating-point number is sometimes estimated (incorrectly) as the relative value of its lowest-order digit. For example, in FPF(16,6) a short hexadecimal floating-point number has six hexadecimal digits, so the relative value of its rightmost digit is

$16^{-6} = 0.000000059604644775390625 \approx 5.96\times10^{-8}$

This has sometimes led people to say that a short hexadecimal floating-point number can accurately represent seven decimal digits. However, the precision of the low-order digit may be smaller:

$16^{-5} = 0.00000095367431640625 \approx 9.54\times10^{-7}$

It's much safer to say that only six decimal digits can be accurately represented.

The question “How many decimal digits does this floating-point number represent?” also involves issues of converting between decimal and the floating-point radix; we'll discuss this topic more fully in Section 36.

Exercises

32.8.1.(2) Estimate the largest relative size of an ulp in each floating-point representation:

1. FPF(16,14)
2. FPF(2,24)
3. FPF(10,34)

[209] It's more usual, and somewhat more correct, to define

```
ulp(x) = min[ successor(x)-x , x-predecessor(x) ]
```

where the predecessor is the number with the next smaller value.

[210] This effect is sometimes called “wobbling precision”.

32.9. Floating-Point Range

We will use positive numbers in the FPF(10,4) representation to illustrate limits on the size of floating-point data. It may help to refer occasionally to Figure 369 on page 583, where the results of the following examples are indicated by "keys" like this: **1**

We will also use [exponent‖fraction] to represent a floating-point value, where the exponent is a signed decimal digit. We'll call the maximum allowed exponent "Emax", and the minimum allowed exponent "Emin". In these examples, Emax=+9 and Emin=−9. This means that the largest normalized value is [+9‖+.9999] (called "Max" or "MaxReal" **6**), and the smallest normalized value is [−9‖+.1000] (called "Min" or "MinReal" **7**).

Floating-point numbers with too-large exponents can't be represented. These exponent range violations are called *exponent overflow* (the exponent is positive, and its value is greater than Emax) and *exponent underflow* (the exponent is negative, and its value is less than Emin).

1. The very large number [+9‖+.5000] multiplied by itself gives [+18‖+.2500], which generates an exponent overflow **1** because the exponent +18 is greater than Emax.
2. Similarly, the very small number [−9‖−.5000] multiplied by itself gives [−18‖+.2500], which generates an exponent underflow because the exponent is less than Emin. **2**

   These two examples show that products and quotients of "normal" numbers can generate results with double the allowed exponent range.
3. Adding the very large number [+9‖+.9000] to itself generates [+10‖+.1800]. The result has an exponent one larger than Emax, which shows that the exponents of sums of "normal" numbers can overflow Emax by 1. **3**
4. Subtracting the very small number [−9‖+.1000] from its next larger value [−9‖+.1001] generates [−12‖+.1000]. Because our precision "p" is 4, this example shows that differences of "normal" numbers can generate exponents as small as Emin−(p−1). **4**
5. Although this normalized result has caused exponent underflow, some floating-point systems support operations that generate unnormalized results for some arithmetic operations. If [−12‖+.1000] is denormalized by shifting the fraction right three places and adding 3 to the exponent, the resulting value is [−9‖+.0001]. **5**

   This is the smallest nonzero value in our FP(10,4) representation. This denormalized minimum value is sometimes called "DMin".

   Multiplying [−5‖+.5000] by itself generates [−10‖+.2500]. The normalized result has created an exponent underflow, but denormalizing the fraction by one digit creates the representable value [−9‖+.0250].

   The two values [−9‖+.0001] and [−9‖+.0250] have valid exponents but unnormalized fractions; they are sometimes called "denormal" or "subnormal" numbers.

Exponent overflow and underflow are serious conditions, and it's important to handle them carefully. Operations on overflowed values are difficult to manage. Sometimes the exponent of the result has been adjusted to lie between Emin and Emax, so you must be careful not to continue calculations with the adjusted value.

Sometimes overflowed results may be set to values like MaxReal, MinReal, a bit pattern representing infinity, or zero, and underflows may be set to zero.

MaxReal is sometimes mistakenly called "infinity", but MaxReal is very different from the mathematical concept of ∞ :

- $\infty / 2 = \infty$, but MaxReal/2 is finite
- $\infty \times 0$ is meaningless, but MaxReal×0 = 0

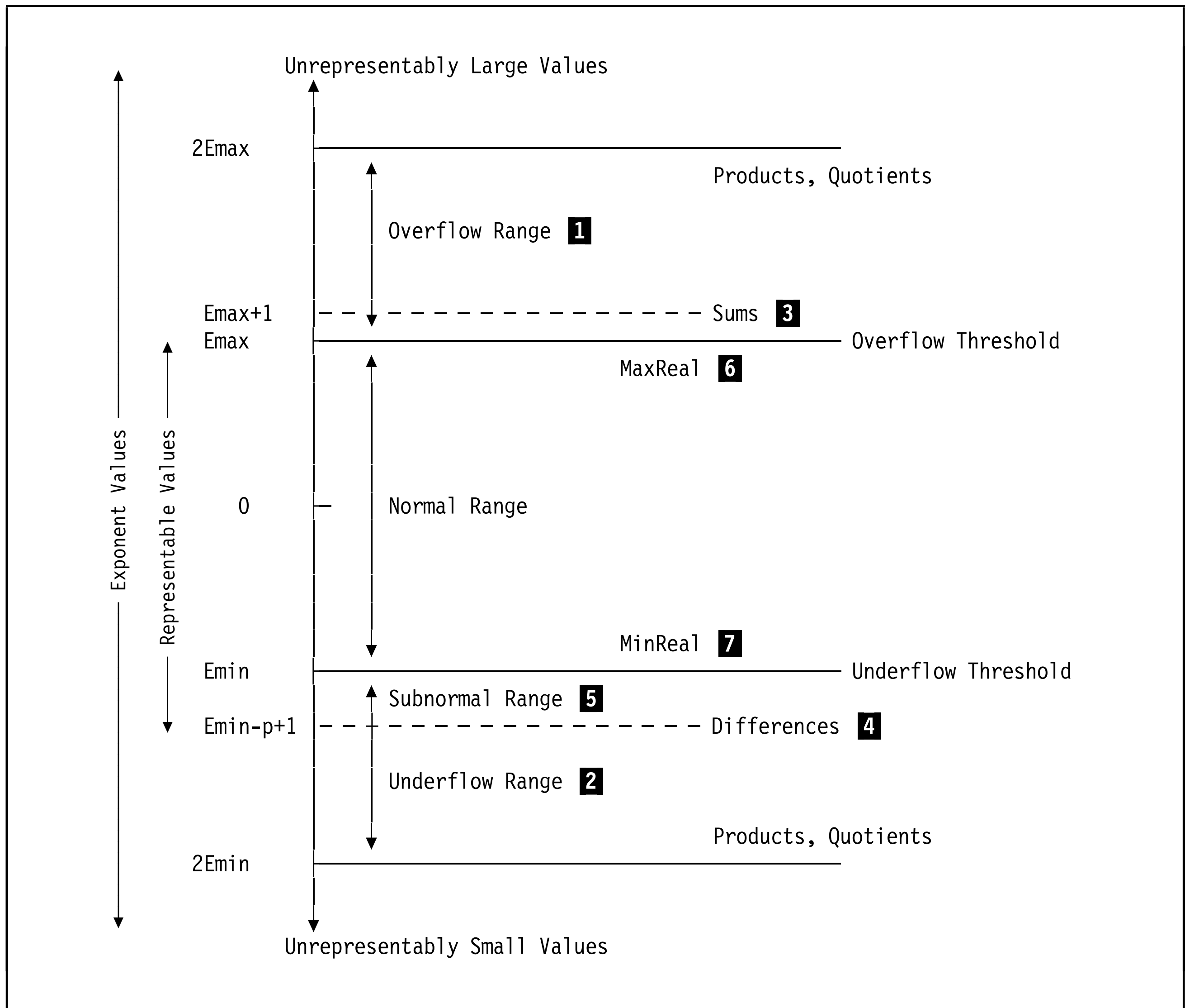


Figure 369. Exponent range of representable and computable values

Exercises

32.9.1.(2) Using our FPF(10,4) representation with a single signed decimal digit for exponent, create three pairs of numbers such that dividing the first number by the second generates

1. an overflowed result with exponent 2×EMax,
2. an underflowed result with exponent 2×Emin, and
3. an underflowed result that can be denormalized to have exponent Emin.

32.9.2.(3) Show the results of these operations in FPF(10,4), and indicate whether the result is normal, overflowed, underflowed, or can be denormalized to have exponent EMin. Assume a single signed decimal digit for an exponent, and that both guard and rounding digits are used.

1. [+8‖+.4946] + [+8‖+.5429]
2. [+8‖+.4946] − [+8‖+.5429]

32.10. Exponents and Characteristics

The examples above used exponents with a sign and a single decimal digit having values from −9 to +9. If the sign and digit were represented in binary on a “real” computer, we would need one bit for the sign and 4 bits for the digit. But with the same 5 bits, we can represent signed *binary* numbers from −16 to +15. This greater range of exponent values shows why floating-point data representations use binary exponents.

All System z floating-point systems use a form of exponent representation called a characteristic. Because exponents can take both positive and negative values, internal arithmetic involving the exponent is simplified by adding a *bias* that makes the characteristic an unsigned number.

This eliminates the problems of dealing with two different signed values in a single number (the significand's + and the exponent's −) and makes it easier to handle and detect both normal and abnormal conditions. When we multiply two numbers, one bias value is subtracted and then the characteristics are added; if a carry is detected after the characteristic addition, we know that an exponent overflow has occurred, and if the result would be negative we know that an exponent underflow has occurred.

Similarly, in division we add one bias value and subtract the characteristic of the divisor; and for addition and subtraction, the difference in characteristics determines the number of places the smaller operand must be shifted.

To illustrate, suppose we decide to restrict our decimal exponent to a single *unsigned* digit. If we choose Emax = +4 and Emin = −5, we need to add a +5 bias for unsigned characteristic values between 0 and 9. The number 100 previously represented as [+3‖+.1000] in this new FPFa(10,4) representation would be [+8‖+.1000]. Similarly, the true MaxReal [+9‖+.9999] would be represented as [+4‖+.9999] and the true MinReal [0‖+.1000] would be represented as [−5‖+.1000].

If we multiply this proposed MaxReal by itself, the characteristic of the intermediate result would be +18−5, or +13, corresponding to an exponent +8. Because Emax = +4, an exponent overflow condition would be recognized. Similarly, if we add this MaxReal to itself, the intermediate result would be [+10‖+.1000] because the rounded fraction was shifted right one position and the characteristic was incremented by one. The characteristic +10 corresponds to an exponent +5, which exceeds Emax by 1.

The choice of a bias value not only determines the range of representable values, but influences arithmetic using those values. For example, suppose we divide 1 by this proposed MinReal: if we use true exponents for a moment, we find

```
[+1‖+.1000] ÷ [-5‖+.1000] = [+7‖+.1000]
```

But this number is about 100 times larger than MaxReal!

The solution to this “range asymmetry” is to choose the bias to be 4 instead of 5. Now, in this updated FPFb(10,4) representation, +MaxReal would be [+5‖+.9999] and +MinReal would be [−4‖+.1000]. Now, if we divide 1 by the new MinReal, the result (again using true exponents) is

```
[+1‖+.1000] ÷ [-4‖+.1000] = [+6‖+.1000]
```

and this result is just a tiny bit larger than MaxReal.

These examples show how the choice of bias can influence the range of computable results. The bias is usually chosen to try to minimize range asymmetries.

Exercises

32.10.1.(2) Short hexadecimal floating-point is a FPF(16,6) representation with EMax=+63 and EMin=−64, with characteristic bias +64. Show the hexadecimal values of MaxReal and MinReal.

32.10.2.(2) Using the same representation as in Exercise 32.9.1, estimate its range asymmetry.

32.10.3.(3) Review Exercise 32.9.1. Then, estimate the range asymmetry if the same exponent range is used with characteristic bias +65. Also, estimate the values of MaxReal and MinReal.

32.11. Summary

This has been a brief overview of floating-point arithmetic. The examples we've seen represent the behavior of the three floating-point representations used in System z, although each has its variations on the themes described in this section.

Terms and Definitions

bias
A fixed value added to an exponent so that the exponent field always contains a nonnegative value, the characteristic.

characteristic
The true exponent plus the bias.

denormalization
A process of shifting the result fraction of a floating-point number to the right by enough digit positions so the exponent will lie in a desired representable range.

exponent overflow
A condition arising when the exponent of a calculated result is too large to be contained in its floating-point representation.

exponent underflow
A condition arising when the exponent of a calculated result is too small to be contained in its floating-point representation.

guard digit
An extra internal digit used to increase the accuracy of a calculated floating-point result.

MaxReal
The largest representable floating-point magnitude, also called Max.

MinReal
The smallest representable floating-point magnitude. If normalized, it is also called Min; if denormalized, it is also called DMin.

normalization
A process of ensuring that the most significant digit in a fraction-based floating-point representation is nonzero.

pre-normalization
A process of normalizing the operand or operands before operating on it or them.

post-normalization
A process of normalizing the result operand after operating on it.

rounding digit
An extra internal digit used to help correctly round a calculated floating-point result.

ulp
An abbreviation for "unit in the last place", a measure of the relative precision of a floating-point number.

33. Hexadecimal Floating-Point Data and Operations

```
 3333333333    3333333333
333333333333  333333333333
33        33  33        33
          33            33
          33            33
       3333          3333
       3333          3333
          33            33
          33            33
33        33  33        33
333333333333  333333333333
 3333333333    3333333333
```

This section examines the hexadecimal floating-point ("HFP") representation and the instructions for hexadecimal floating-point arithmetic. Hexadecimal floating-point was introduced with the original System/360, and is the oldest of the three System z floating-point representations.

33.1. Hexadecimal Floating-Point Data

The hexadecimal floating-point representation uses hexadecimal fraction digits and a binary exponent representing a power of 16. All significant digits are present; none are implied. A number X in hexadecimal floating-point form is

$$X = f \times 16^{e}$$

where "f" is a fraction with value less than one, and "e" is a positive or negative binary integer exponent with value between −64 and +63, so that

$$-64 \le e \le +63$$

For example, the number 1 can be represented in decimal as 100×10^{-2}, 1.00×10^{0}, and 0.0001×10^{4}, and in hexadecimal as `X'100'`$\times 16^{-2}$, `X'1'`$\times 16^{0}$, and `X'0.0001'`$\times 16^{4}$. The normalized short floating-point form is `X'.100000'`$\times 16^{1}$.

Figure 370 on page 587 shows the three hexadecimal floating-point formats. The sign bit S is 0 for positive numbers and 1 for negative numbers. The exponent is represented as an unsigned characteristic in two's complement form, obtained by adding a bias 64 to the exponent:

```
characteristic = exponent + 64 = exponent + X'40'
```

Thus, the characteristic satisfies

$$0 \le \text{characteristic} \le 127$$

The sign and seven-bit characteristic occupy the first byte of the number, and the magnitude of the fraction is carried in the remaining bytes, as shown in Figure 370 on page 587.

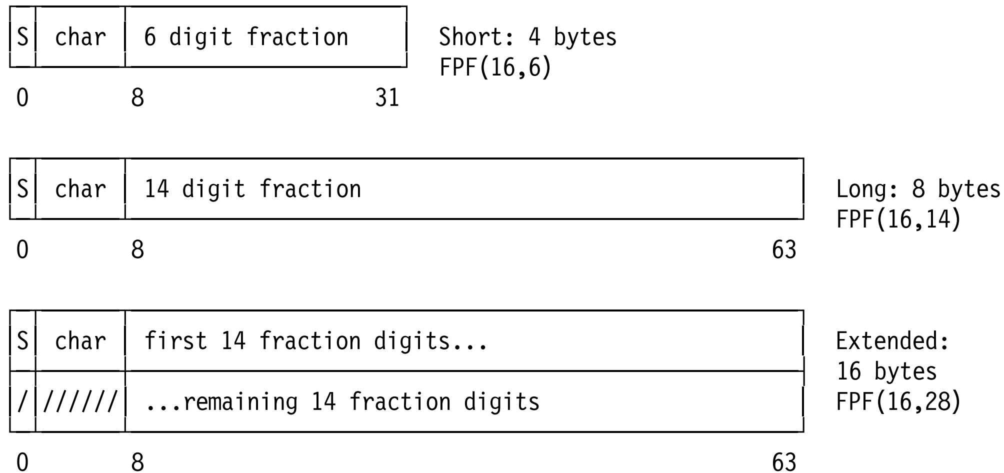


Figure 370. Hexadecimal floating-point number representations

The short-format fraction has 6 hexadecimal digits, the long-format fraction has 14, and the extended-format fraction has 28. The high-order byte of the second portion of the extended format is ignored.[211]

To illustrate a short hexadecimal floating-point representation, consider the value +123.4567 that we converted to scaled fixed-point binary in Section 31.3.

1. The integer value 123 = `X'7B'`, giving the first two digits of the hexadecimal fraction. Because the significand is a fraction, 123 must be represented as `X'.7B'`$\times 16^{+2}$. Thus, we know that the exponent is +2 and the characteristic is 66 = `X'42'`.
2. The fraction value .4567 = `X'.74EA48A...'`, which rounds to `X'.74EA'`.
3. We now combine the parts:
   - The sign is +, so the first bit is zero.
   - The characteristic is `X'42'`
   - The hexadecimal fraction is `X'.7B74EA'`
   - The converted representation is `X'427B74EA'`

Table 228 summarizes the properties of hexadecimal floating-point numbers. The column headed "Precision" gives the number of hexadecimal digits in the fraction.

Byte length	Char. (bits)	Min exp.	Max exp.	Char. Bias	Pre-cision	Max Norm. (Max)	Min Norm. (Min)	Min Denorm. (DMin)
4	7	−64	+63	+64	6	$7.2\times 10^{+75}$	5.4×10^{-79}	5.1×10^{-85}
8	7	−64	+63	+64	14	$7.2\times 10^{+75}$	5.4×10^{-79}	1.2×10^{-94}
16	7	−64	+63	+64	28	$7.2\times 10^{+75}$	5.4×10^{-79}	1.7×10^{-111}

Table 228. Hexadecimal floating-point data representations

These rules determine a unique representation for a normalized number X:

1. If f = 0, and e = −64, then X is a *true zero*; that is, everything except possibly the sign bit S is zero.
2. If f = 0 and e > −64, then X is a *pseudo-zero* (the characteristic is nonzero).
3. If 1.0 > f ≥ 1/16, then X is said to be *normalized* (the most significant fraction digit is nonzero).

[211] The unused bits of the extended format were used to greatly extend the exponent range in a special form called the "Extended Exponent Range Representation", or "XEXP". It was implemented only in software.

However, the representation of X could be unnormalized:

4. If 1/16 > f > 0, then X is *unnormalized* (there is at least one leading zero digit in the fraction).

A number X has a unique *normalized* representation, and may have several unnormalized representations with the same value. For example, we can represent the number 1.0 in several ways; only the first is normalized:

```
41100000    42010000    43001000    44000100    45000010    46000001
```

If the value is not an integer, the unnormalized forms may not have the same value as the normalized form. For example, if we chose to represent the decimal fraction 0.1 as an unnormalized short floating-point number, it might appear to have the following equivalent representations:

```
4019999A    4101999A    4200199A    4300019A    4400001A    45000002
```

The values represented become less accurate as the amount of unnormalization increases.

```
 4019999A    4101999A    4200199A    4300019A    4400001A    45000002
.100000023  .100000381  .100006104  .100097656  .101562500  .125000000
```

In the last case, the value of the number is not 0.1 (base 10) but 0.125, which would be unacceptably different for most applications. Thus, normalized floating-point numbers offer the greatest *precision*: they contain more digits with reliable values. This is not the same as saying that normalized results are more *accurate*, because there may be other errors in a calculation which cause the results to be incorrect even though all six digits are always retained in the answer.

Table 229 shows some unnormalized numbers in short hexadecimal floating-point format, and their corresponding normalized forms:

Unnormalized	Normalized
40030000	3F300000
41003000	3F300000
620AB128	61AB1280
4300009E	3F9E0000
CB00FACE	C9FACE00
03000123	00123000

Table 229. Unnormalized and normalized short hexadecimal floating-point numbers

Table 230 shows some examples of short hexadecimal floating-point numbers: Their representations are fairly simple because all the quantities are multiples of powers of two.

Value	Fraction-exponent representation	Short floating-point representation
1	$+1\times16^{1}$	41100000
1	$+.0001\times16^{4}$	44000100
256	$+.000001\times16^{8}$	48000001
7/8	$+E\times16^{0}$	40E00000
−5/4	$-.14\times16^{1}$	C1140000
1/1024	$+.4\times16^{-2}$	3E400000
1000/1024	$+.FA\times16^{0}$	40FA0000

Table 230. Short hexadecimal floating-point numbers

Consider the representation of +1000/1024:

- First, $1000._{10}$ = `X'3E8.' = B'0011 1110 1000.'`
- Dividing by 1024 shifts this value right 10 bits, giving `X'.FA' = B'.1111 1010 0000'`

- The result has exponent 0 and characteristic X'40', giving X'40FA0000' as the hexadecimal floating-point representation.

Long floating-point numbers are 8 bytes long; Table 231 shows some typical examples.

Value	Fraction-exponent representation	Long floating-point representation
1	$+1\times16^{1}$	41100000 00000000
0.1	$+.1\times10^{-1}$	40199999 9999999A
−256	-1×16^{3}	C3100000 00000000
1000	$+1000.0\times10^{3}$	433E8000 00000000
7/8	$+E\times16^{0}$	40E00000 00000000
−5/4	$-.14\times16^{1}$	C1140000 00000000
1/1024	$+.4\times16^{-2}$	3E400000 00000000
1000/1024	$+.FA\times16^{0}$	40FA0000 00000000

Table 231. Long hexadecimal floating-point numbers

An extended-precision number is 16 bytes long, and is represented by a pair of long-precision numbers, the high-order and low-order halves. The high-order half contains the sign, the characteristic, and the most significant 14 hex digits of the fraction of the extended number. The sign and characteristic of the low-order half are ignored, and the fraction part of the low-order half contains the least significant 14 digits of the fraction.[212] (By convention, the characteristic of the low-order half is set to 14 less than the characteristic of the high-order half.) Thus, an extended number has a precision of 28 hexadecimal digits, equivalent to about 32 decimal digits.

Table 232 shows some examples of extended hexadecimal floating-point numbers.

Value	Fraction-exponent representation	Extended floating-point representation
0.1	0.1×10^{0}	40199999 99999999 32999999 9999999A
π	$.314159...\times10^{+1}$	413243F6 A8885A30 338D3131 98A2E037
e	$.271828...\times10^{+1}$	412B7E15 1628AED2 33A6ABF7 158809CF
.000000000001	$.1\times10^{-11}$	37119799 812DEA11 29197F27 F0F6E886

Table 232. Extended hexadecimal floating-point numbers

Exercises

33.1.1.(2) A four-byte area of memory contains the bit pattern X'4040405C'. What is represented by that pattern? (You should now be able to describe five different possibilities.)

33.1.2.(2) How many redundant values are there for these short HFP numbers? (a) 25, (b) 1, (c) 0.25, (d) 0. How many redundant values for *long* numbers?

33.1.3.(1)+ 25.5. Determine if each of the following seven short HFP numbers is a pseudo-zero, a true zero, is normalized, or is unnormalized.

1. X'45678900'
2. X'FFFFFFFF'
3. X'00000001'
4. X'80000000'

---

[212] The two halves of an extended-precision hexadecimal floating-point number are *not* treated as independent values, as is done on some other systems supporting “double-double” precision.

5. X'00FF00FF'
6. X'40000000'
7. X'C400C1D4'

33.1.4.(3) The “XEXP” representation mentioned in the footnote on page 587 stated that the unused bits of the low-order characteristic were used to extend the exponent range of extended-precision numbers. Assuming that the sign bit was reserved and the remaining 7 bits were used to extend the high-order characteristic, what would be the resulting exponent range?

33.1.5.(2)+ Given the quantities Z, A, B, C, D, E, and D as in Exercise 2.8.5, give their representations in short hexadecimal floating-point form.

33.1.6.(2) Given a short precision hexadecimal floating-point number S, write fixed-point instructions to calculate short floating-point numbers with (a) the largest power of 16 ≤ S, and (2) the smallest power of 16 ≥ S. Store the results at S1 and S2. Ignore the possibility of exponent underflow or overflow.

For example, if S=X'42280000'=40_{10}, then the result at S1 is X'42100000'=16^1 and the result at S2 is X'43100000'=16^2.

33.1.7.(2) Can you extend the instructions you wrote in solving Exercise 33.1.6 for long and extended precision operands? Why or why not?

33.2. Writing Hexadecimal Floating-Point Constants

As mentioned in Section 31.8, the basic lengths of all floating-point constants are determined by types E (short), D (long), and L (extended). The default length and alignment are word, doubleword, and doubleword. The default alignment in memory of an extended-precision number is only to a doubleword boundary, not to a 16-byte “double-doubleword” or “quadword” boundary.

The syntax of DC assembler instruction statements for HFP constants is

```
         DC    [dup_factor]type[modifiers]'value[dec_exponent]'
```

Like many other numeric constants, you can provide several value[dec_exponent] values separated by commas.

where

- the duplication factor [dup_factor] is optional,
- the type is required,
- the [modifiers] are optional,
- a value is required, and
- the decimal exponent [dec_exponent] is optional.

This constant has all five items:

```
         DC    2EL4S1E-2'7.73E8'
```

The duplication factor is 2, the type is E, the three modifiers are “L4“, “S1”, and “E−2”;, the value is 7.73, and the decimal exponent is E8.

Table 233 on page 591 gives some examples of simple floating-point constants. For extended-precision constants, the Assembler gives the same signs to the high-order and low-order halves, and the characteristic of the low-order half is 14 less than the characteristic of the high-order half. The CPU does the same for the results of extended-precision arithmetic operations.

DC Operand	Assembled Constant
E'1000'	433E8000
E'22.75'	4216C000
D'3.142'	413245A1 CAC08312
E'100.04'	42640A3D
D'.0000923'	3D60C897 B3E64BFA
D'729'	432D9000 00000000
E'-55'	C3370000
D'-7088.263'	C41BB043 53F7CED9
L'1.0'	41100000 00000000 33000000 00000000
L'0.1E-70'	0611AB20 E472914A 786BEAF3 890FCB47

Table 233. Assembled hexadecimal floating-point constants

Consider the second constant: 22 = X'16', and .75 = 12/16 = X'.C'. Putting these two parts together we get 22.75 = X'16.C' or X'.16C'$\times16^{+2}$. (Remember, the significand is represented as a hexadecimal fraction!) So the characteristic is +2+64= X'42', and the entire 8-digit constant is therefore X'4216C0000'.

Then consider the last constant: the characteristic of the second half is much *larger* than the characteristic of the first half. This is because the low-order characteristic is the high-order characteristic -X'14': X'06'-X'14' (modulo 128), or X'78'.

The Assembler's decimal to hexadecimal floating-point conversion is rounded, but hexadecimal floating-point arithmetic performed by the CPU is almost always unrounded.

33.2.1. Decimal Exponents

If a constant has many leading or trailing zeros, it is inconvenient to have to write out all the digits; the operand E'10000000000000' is an awkward way to define the constant 10^{13}. To simplify writing such constants, two different methods may be used, both of which specify a power of ten by which the value must be multiplied: one is to use a *decimal exponent* with the nominal value, and the other is to use an *exponent modifier*.

As we saw for F and H-type binary constants in "12.1. F-Type and H-Type Constants" on page 146, a decimal exponent is written as the letter "E" followed by a positive or negative integer which specifies the power of ten by which the constant is to be multiplied. It is written immediately following the value of the constant. For instance, we could write a DC statement operand to define four constants of value 10^{13} like this:

```
         DC    E'1E13',E'1.0E13',E'100E11',E'10.E+12'
```

Table 234 shows some examples of constants with decimal exponents.

DC Operand	Assembled Constant
D'.1E+4'	433E8000 00000000
D'.1E+3'	42640000 00000000
D'.1E-1'	3F28F5C2 8F5C28F6
D'.1E-2'	3E418937 4BC6A7F0
D'.1E-3'	3D68DB8B AC710CB3
D'.1E-4'	3CA7C5AC 471B4784

Table 234. Hex floating-point constants with decimal exponents

The previous rules for using literals also apply to hexadecimal floating-point literals.

Exercises

33.2.1.(1) Some HFP constants have one or more trailing zero digits. Does the generated constant for

```
         DC    E'1E13'
```

contain all the significant bits of the value 10^{13}? How could you tell whether it does or does not?

33.2.2.(2) Write a single DC statement that generates a table of 11 short HFP constants containing the powers of 10 from 0 to 10.

33.2.3.(2) What is the largest power of 10 that can be stored precisely as (1) a short, (2) a long, (3) an extended HFP number?

33.2.4.(1)+ Write statements to generate the nine hexadecimal floating-point constants for the decimal fractions from 0.1 to 0.9 in steps of 0.1.

33.3. Modifiers

Hexadecimal floating-point constants allow three types of modifiers: Length, Scale, and Exponent. If any are present, they *must* appear in that order.

33.3.1. Length Modifiers

As with other constants, you can use the L modifier to define explicit lengths; the generated constants are not aligned. Truncation or padding takes place on the right. Thus, the DC operand EL2'1' generates the constant X'4110', and DL3'0.1' generates X'40199A'. The Assembler rounds the fraction to the specified length.

Length modifiers are rarely used with floating-point constants, and then mainly to cause unaligned data in a tightly-packed data structure. Generating a shorter constant loses precision, as Table 235 shows:

DC Operand	Generated Constant
EL4'2.71965'	X'412B83B0'
EL3'2.71965'	X'412B84'
EL2'2.71965'	X'412C'
EL1'2.71965'	X'41' (Error!)

Table 235. Length-modified hexadecimal floating-point constants

Note that as each fraction grows shorter, the lowest-order digit is rounded to compensate for the truncated portion that was removed. The Assembler issues an error message for the last case to indicate that all precision has been lost.

33.3.2. Scale Modifiers (*)

Obscure topic

Because scale modifiers are used very rarely for floating-point constants, feel free to skip this subsection.

Like the scale modifiers described in Section 31.3.2 for binary constants, a scale modifier is used to generate an unnormalized HFP constant. It is written as the letter "S" followed by either a decimal self-defining term or a nonnegative absolute expression enclosed in parentheses; the decimal value or the parenthesized expression may be preceded by a "+" sign if desired. The

value of the scale modifier determines the number of leading zero digits in the fraction: the number of hex digit positions the fraction is to be shifted to the right.

Scale modifiers larger than 5 for short, 13 for long, and 27 for extended operands will cause all significant digits to be lost, so the Assembler considers this an error.[213] The characteristic of the generated number is adjusted upward by the number of hex digits shifted, so that the value of the constant remains the same, subject to the possible loss of accuracy caused by the right shift. For example:

```
DC    ES0'1.0'          Generates X'41100000'
DC    ES1'1.0'          Generates X'42010000'
DC    ES2'1.0'          Generates X'43001000'
DC    ES3'1.0'          Generates X'44000100'
DC    ES4'1.0'          Generates X'45000010'
DC    ES5'1.0'          Generates X'46000001'
DC    ES5'15.999'       Generates X'47000001'  (rounded!)
```

The Assembler rounds the remaining portion of the fraction to account for the part that was shifted off; if after rounding there is a carry into a zeroed digit position, the fraction is shifted right once more, and the characteristic is adjusted accordingly.

33.3.3. Exponent Modifiers

Unlike a decimal exponent, an exponent modifier appears *before* the nominal value of the constant or constants. It is written as the letter E followed by either a decimal self-defining term or an absolute expression enclosed in parentheses. A sign may precede the decimal value or the parenthesized expression.

We can write the constant 10^{13} as E**E13**'1', E**E6**'100E5', E**E-2**'1.0E15', E**E-(3-1)**'1.E+15' or E**E17**'1E-4', where the exponent modifier is indicated by a bold-face **E** and the decimal exponent by an underscored E, as in E15. The modifying power of ten applied to each constant is the sum of the decimal exponent and the exponent modifier.

Either method is accepted for a given constant, so you might ask why both methods are used. One possibility is that you might want to specify multiple constants in a single operand, as in

```
DC    EE2'1,20,3E3,4E22'     Four short HFP constants
```

where the exponent *modifier* 2 applies to all the constants generated, while each decimal *exponent* affects only the value to which it is appended. The values of the third and fourth operands are affected by both an exponent modifier and a decimal exponent.

The allowed range for exponents is such that the sum of the decimal exponent and the exponent modifier must lie between −85 and +75.

The constants in Table 236 on page 594 will be assembled as shown, where we assume the absolute expression (B−A) has value +4.

As noted following Table 235 on page 592, the characteristic is adjusted to compensate for shifts due to scaling.

213 If the scale modifier is equal to the fraction length, the assembler notes that precision is lost; if the modifier exceeds the fraction length, it is considered invalid.

DC Operand	Assembled Constant
EL5E1'10'	4264000000
EL3S1E1'10'	430640
DS13'.5'	4E000000 00000008
ES1E1'1'	420A0000
E'.999999'	40FFFFEF
EE-((8+4)/6)'1E15'	4B9184E7
EL(B-A)S(B-A)E(B-A)'4'	4800009C
ES5'31'	47000002
2EL4S1E-2'7.73E8'	47075F3547075F35

Table 236. Hexadecimal floating-point constants with modifiers

Exercises

33.3.1.(2) What would you expect the Assembler to generate for these hexadecimal floating-point constants?

```
A        DC    EL1'25'
B        DC    EL2'.1'
C        DC    DL2'999'
D        DC    DL1'1'
E        DC    EL2'1000'
```

33.3.2.(1) What decimal values are represented by these constants?

```
W        DC    DE5'3.7E-2'
X        DC    EE-25'1.0E80'
Y        DC    LE2'8.8E-2'
Z        DC    EE9'4E9'
```

33.4. Subtypes Q and H (*)

Hexadecimal floating-point constants allow two subtypes, Q and H.

33.4.1. LQ-Type Constants

The Q subtype is used only with type L, and its only effect is to align the generated constant on a quadword (16-byte) boundary. For example:

```
DC    L'0.1'              Doubleword aligned
DC    LQ'0.1'             Quadword aligned
- - -
DS    0LQ                 Align to quadword boundary
DC    CL32'Characters!'   Quadword-aligned character string
```

Figure 371. Quadword aligned constants and data

To be sure your constants and data are correctly aligned on quadword boundaries when your program is loaded into memory for execution,

1. use the Assembler's SECTALGN(16) option to request that all control sections[214] begin on quadword boundaries, and

[214] More about control sections in Section 38.

2. verify that the linker and loader of your operating system support quadword alignment.

33.4.2. Subtype H

Specifying subtype H for hexadecimal floating-point constants lets you specify

- more precise rounding
- value suffixes to choose a rounding mode
- symbolic operands (Max), (Min), and (DMin) with optional signs

Without the H subtype, the Assembler rounds HFP constants by adding 1 to the first “lost” bit position, just as we do decimal rounding by adding 5 to the first “lost” decimal digit. When you specify subtype H, five rounding modes can be specified by adding “Rn” at the end of the nominal value, where the number “n” selects a rounding mode shown in Table 237. mode shown in Table 237.

Mode	Rounding
R1	Add 1 to the first lost bit (this is the default mode).
R4	Round the exact value to the nearest representable machine value. If the exact value is exactly half-way between two machine numbers, choose the one with a zero low-order bit.
R5	Round toward zero: truncate by discarding any extra bits.
R6	Round toward the maximum positive value (“toward $+\infty$”)
R7	Round toward the maximum negative value (“toward $-\infty$”)

Table 237. Hexadecimal floating-point rounding modes with subtype H

The rounding-mode suffix must follow the value of the constant, including any decimal exponent.

Figure 372 gives some examples of HFP constants using rounding-mode suffixes.

```
DC    EH'0.1R1'       generates X'4019999A'
DC    EH'0.1R4'                 X'4019999A'
DC    EH'0.1R5'                 X'40199999'
DC    EH'0.1R6'                 X'4019999A'
DC    EH'0.1R7'                 X'40199999'
DC    EH'1048576.5R1'           X'46100001'  rounded up
DC    EH'1048576.5R4'           X'46100000'  rounded down to even
DC    EH'1048577.5R1'           X'46100002'  rounded up
DC    EH'1048577.5R4'           X'46100002'  rounded up to even
```

Figure 372. Hexadecimal floating-point constants with rounding suffixes

The value 1048576.5 has hexadecimal representation X'100000.8', exactly halfway between X'100000' and X'100001'. Standard rounding (R1) rounds the “lost” bit up, while “Round to Even” (R4) rounds to the nearest number with a low-order zero bit, X'100000'. Similarly, the representation of 1048577.5 is X'100001.8'; in this case, “Round to Even” rounds up to X'100002'.

The difference between these two rounding modes is sometimes important when you create data for floating-point arithmetic.

Table 238 on page 596 shows the symbolic operands and their generated constants.

Constant Type	DC Operand	Generated Constant
Maximum Magnitude	EH'(Max)' DH'(Max)' LH'(Max)'	X'7FFFFFFF' X'7FFFFFFF FFFFFFFF' X'7FFFFFFF FFFFFFFF 71FFFFFF FFFFFFFF'
Minimum Normalized Magnitude	EH'(Min)' DH'(Min)' LH'(Min)'	X'00100000' X'00100000 00000000' X'00100000 00000000 72000000 00000000'
Minimum Denormalized Magnitude	EH'(DMin)' DH'(DMin)' LH'(DMin)'	X'00000001' X'00000000 00000001' X'00000000 00000000 72000000 00000001'

Table 238. Symbolic hexadecimal floating-point constants

33.4.3. Difficult Numbers (*)

Obscure topic

This section will be interesting mainly to mathematically inclined readers.

In addition to symbolic operands and rounding modes, H subtypes are converted using greater internal precision, so certain "difficult" numbers may be slightly more accurate.

"Difficult" numbers are those whose exact values lie very close to half way between two representable numbers, so extra precision is needed to determine how they should be rounded. Not only are they difficult to convert accurately, it's difficult to find them! Here are some examples:

DC Operand	Without H Subtype	With H Subtype
D'.303325544866797714604E-10'	X'382159DA E5B7B6BD'	X'382159DA E5B7B6BE'
D'.185240322463448422373E-23'	X'2D23D4A8 0F402692'	X'2D23D4A8 0F402693'
E'.1053771313464019060319004056804E-41'	X'1E177FF8'	X'1E177FF9'
L'.8031692147E-10'	X'38584F34 1F25338E 2A9D527E 34864A16'	X'38584F34 1F25338E 2A9D527E 34864A17'

Table 239. "Difficult" hexadecimal floating-point conversion values

Exercises

33.4.1.(5) Find another "difficult" number.

33.4.2.(2) Write and assemble the third (E-type) difficult number as an L-type constant. What property does the generated constant have that makes it "difficult"?

33.4.3.(2) The hexadecimal floating-point value of 10^{-3} to many digits is

```
3E418937 4BC6A7EF 9DB22D0E 56041893 74BC6A7E F9DB22D0 E5604189 ...
```

If this is rounded to long format using rounding-mode suffixes R1, R4, and R5, what will be the generated results?

33.4.4.(5) Choose a number from Table 239 and try to generate its value to a high enough precision, to understand why it's "difficult" to round correctly.

33.5. Basic Hexadecimal Floating-Point Instructions

We will be concerned mainly with instructions for multiplying, dividing, adding, and subtracting hexadecimal floating-point operands. We'll describe some details of their operation, but you don't need to fully understand these details to use them.

For these instructions, the CPU

- uses a single hexadecimal guard digit, and
- truncates all results without rounding.

Later in this section we'll see some instructions that round their results.

In most operations on extended-precision numbers, the CPU gives the same signs to the high-order and low-order halves, and the characteristic of the low-order half will be 14 less than the characteristic of the high-order half.

33.6. Hexadecimal Floating-Point RR-Type Data-Movement Instructions

Some instructions used for all types of floating-point operands were described in Section 31.9. The instructions in Table 240 are specifically for hexadecimal operands.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
32	LTER	RR	Load and Test (Short)	33	LCER	RR	Load Complement (Short)
31	LNER	RR	Load Negative (Short)	30	LPER	RR	Load Positive (Short)
22	LTDR	RR	Load and Test (Long)	23	LCDR	RR	Load Complement (Long)
21	LNDR	RR	Load Negative (Long)	20	LPDR	RR	Load Positive (Long)
B362	LTXR	RRE	Load and Test (Extended)	B363	LCXR	RRE	Load Complement (Extended)
B361	LNXR	RRE	Load Negative (Extended)	B360	LPXR	RRE	Load Positive (Extended)

Table 240. Data-moving hexadecimal floating-point instructions

The register-to-register "Load and Test", "Load Positive", "Load Negative", and "Load Complement" instructions are similar to the corresponding general register instructions:

- Load and Test sets the Condition Code to indicate the sign of a value
- Load Positive sets the sign to +
- Load Negative sets the sign to − for nonzero values
- Load Complement inverts the sign for nonzero values.

For hexadecimal floating-point, there are a few differences:

1. Negation only needs to invert the sign bit, and not complement all the other bits. Thus, no overflow is possible when complementing an operand, and the characteristic and fraction are unmodified.
2. In instructions that set the Condition Code, an operand is treated as zero if its fraction part is zero. A pseudo-zero is treated as a zero even though its characteristic is nonzero.
3. If short operands are specified, only the left half of the register is involved. This means that in testing the contents of the result register R_1 in order to set the Condition Code, the low-order half of the register is ignored. For example, if

   ```
   c(FPR2) = X'42000000 12345678'
   ```

   the instructions "LTER 2,2" and "LTDR 2,2" yield CC settings of 0 and 2 respectively.
4. Because only the sign bit is manipulated, it is possible to generate a "minus zero", a number with all zero bits except for sign. It still behaves like a true zero.
5. Operands are not normalized.

The CC settings after floating-point arithmetic are the same as for the GPR instructions: 0 for a zero fraction, 1 for a negative result, and 2 for a positive result. These instructions do not set Condition Code 3.

For example:

```
         LD    0,=DH'(Max)'        Initialize FPR0
         LTDR  0,0                 CC = 2
         LCDR  6,0                 CC = 1
         LPDR  6,6                 CC = 2
*
         LE    4,=X'46000000'      Pseudo-zero in FPR4
         LTER  4,4                 CC = 0, c(FPR4)=X'46000000'
         LCER  2,4                 CC = 0, c(FPR2)=X'C6000000'
```

Figure 373. Examples of hexadecimal floating-point instructions

These extended-precision instructions have an additional behavior: for nonzero fractions, the high-order and low-order signs are made equal, and the low-order characteristic is set to the high-order characteristic minus 14, modulo 128.

```
         LD    4,=X'4A123456789ABCDE'  Initialize FPR4
         LD    6,=X'00EDCBA987654321'  ..and FPR6
         LTXR  0,4           c(FPR0,2)=X'4A123456789ABCDE..
*                                  ..3CEDCBA987654321', CC=2
```

Figure 374. Example of LTXR instruction

If the source operand is a pseudo-zero, the result characteristic and fraction are set to zero, and the sign bits of the high-order and low-order halves are made identical.

```
         LZXR  4                          Set c(FPR4,6) to zero
         LD    4,=X'CE00000000000000'     Negative pseudo-zero in FPR4
         LTXR  0,4           c(FPR0,2)=X'8000000000000000..
*                                  ..8000000000000000', CC=0
         LCXR  0,4           c(FPR0,2)=X'0000000000000000..
*                                  ..0000000000000000', CC=0
```

Figure 375. Examples of extended-precision hexadecimal RR instructions

Exercises

33.6.1.(1) If you execute these instructions:

```
         LD    4,=X'50123456789ABCDE'
         LD    6,=X'FEDCBA9876543210'
         LTXR  0,4
```

What will be in the register pair (FPR0,FPR2), and what will be the CC setting?

33.6.2.(2) Suppose c(FPR0)=X'0A123456789ABCDE' and c(FPR2)=X'42857196DBB93310'. Show the CC setting and the contents of the result register or registers after executing each of the following instructions:

```
(1)      LPER  4,2
(2)      LTDR  2,2
(3)      LCXR  4,0
(4)      LCDR  4,2
```

33.7. Hexadecimal Floating-Point Multiplication

Table 241 lists the hexadecimal floating-point multiplication instructions. None of them affect the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED37	MEE	RXE	Multiply (Short×Short to Short)	B337	MEER	RRE	Multiply (Short×Short to Short)
7C	ME, MDE	RX	Multiply (Short×Short to Long)	3C	MER, MDER	RR	Multiply (Short×Short to Long)
6C	MD	RX	Multiply (Long×Long to Long)	2C	MDR	RR	Multiply (Long×Long to Long)
67	MXD	RX	Multiply (Long×Long to Extended)	27	MXDR	RR	Multiply (Long×Long to Extended)
				26	MXR	RR	Multiply (Ext.×Ext. to Ext.)

Table 241. Hexadecimal floating-point Multiply instructions

Multiplication is the simplest HFP operation. Suppose we need to find the product z of two numbers w and y. From the usual rules of algebra, we know that if $w = F_w \times 16^{Ew}$ and $y = F_y \times 16^{Ey}$ then

$$z = (F_w \times F_y) \times 16^{(Ew+Ey)}$$

so that $F_z = F_w \times F_y$ and $Ez = Ew + Ey$.

The CPU sets the result to a true zero immediately if either fraction is zero. Otherwise, the characteristic C_z of the result is computed from

$$C_z = C_w + C_y - 64 = C_w + C_y - X'40'$$

where X'40' is subtracted so that C_z is biased only once. (See Exercise 33.7.7.)

The two fractions are pre-normalized, and the result characteristic is adjusted internally to account for both shifts. The CPU then multiplies the two fractions to obtain the result fraction F_z. Since F_x and F_y were pre-normalized and both are nonzero, we must have a value for F_z in the range

$$1.0 > F_z \geq 16^{-2}$$

so it may be necessary to shift F_z left one digit to post-normalize the result fraction.

The sign of the result is determined from the rules of algebra, which means that an XOR is performed between the sign bits of the original operands. The result is placed in the floating-point register specified by the first operand of the instruction.

Suppose we multiply $w = 5$ and $y = 7$. Then, C_w = X'41', C_y = X'41', F_w = X'.500000', and F_y = X'.700000'. The result characteristic C_z is

$$C_z = X'41' + X'41' - X'40' = X'42'$$

and the result fraction is X'.230000', which requires no post-normalization. Thus, in short floating-point form, we find that z = X'42230000'. The instructions in Figure 376 do this:

```
         LE    2,W                Put W in FPR2
         MEE   2,Y                Multiply by Y
         STE   2,Product          Store the result
         - - -
W        DC    E'5'               X'41500000'
Y        DC    E'7'               X'41700000'
Product  DS    E                  Result = X'42230000'
```

Figure 376. Short hexadecimal floating-point multiplication

Figure 377 on page 600 shows the contents of the two floating-point registers.

FPR2 Before		Operation	FPR2 After	
////////	////////	LE 2,W	41500000	////////
41500000	////////	MEE 2,Y	42230000	////////

Figure 377. Floating-point registers used for hexadecimal floating-point multiplication

To illustrate a typical use of hexadecimal floating-point multiplication, suppose we must calculate Z(i)=X(i)×X(i)×Y(i) for values of i from 1 to 10, where X(i), Y(i), and Z(i) are short floating-point numbers in tables at **XX**, **YY**, and **ZZ** respectively.

```
         XR    5,5                GR5 contains index
         LH    2,Count            Counter in GR2
Loop     LE    0,XX(5)            Load X(i) in FPR0 (short)
         MEER  0,0                X(i)*X(i) in FPR0
         MEE   0,YY(5)            Multiply by Y(i)
         STE   0,ZZ(5)            Store short result Z(i)
         LA    5,L'XX(,5)         Increment index by operand length
         JCT   2,Loop             Count down and loop
         - - -
XX       DC    E'1,2,3,4,5,6,7,8,9,10' Values of X(i)
Count    DC    Y((*-XX)/L'XX)     Count of X(i) entries
YY       DC    5E'3.14159,2.71828'   2 YY(i) values, copied 5 times
ZZ       DS    10E                Results Z(i)
```

Figure 378. Calculating a table of short hexadecimal floating-point products

Note that we use the length attribute of the symbol **XX** twice: once as an increment in the loop, and once in defining the symbol **Count**. If new entries are added to the X(i) table, or if the X(i) entries are updated to long precision, the value at **Count** will be updated automatically when you reassemble the program.

The general registers provide addressing and indexing; all floating-point arithmetic uses FPR0. If the same calculation is done with *long* operands, the program segment would appear as in Figure 379.

```
         XR    5,5                Initialize index to zero
         LH    2,Count            Number of elements (10)
Loop     LD    0,XX(5)            Load X(i) in FPR0 (long)
         MDR   0,0                X(i)*X(i) in FPR0
         MD    0,YY(5)            Multiply by Y(i)
         STD   0,ZZ(5)            Store long result Z(i)
         LA    5,L'XX(,5)         Increment index by operand length
         JCT   2,Loop             Count down and branch
         - - -
XX       DC    10D'27'            Some values for X(i)
Count    DC    Y((*-XX)/L'XX)     Count of X(i) entries
YY       DC    10D'-263E-17'      And for Y(i)
ZZ       DS    10D                Space for results
```

Figure 379. Calculating a table of long hexadecimal floating-point products

Here, the length attribute of the symbol **XX** is 8; in both figures, the increment in the loop and the number of elements at **Count** are determined automatically by the Assembler.

The hexadecimal floating-point multiply instructions in Table 241 on page 599 generate products of different lengths, as summarized in Table 242 on page 601. None of them set the Condition Code.

Operand 1	Operand 2	Product	Instructions	Result
Short	Short	Short	MEE, MEER	Truncated to 6 digits; right half of FPR R_1 unchanged
Long	Long	Long	MD, MDR	Truncated to 14 digits
Extended	Extended	Extended	MXR	Truncated to 28 digits
Short	Short	Long	MDE (ME), MDER (MER)	Exact product
Long	Long	Extended	MXD, MXDR	Exact product

Table 242. Summary of hexadecimal floating-point multiplication results

MEE and MEER generate the same short product as the high-order four bytes of the product generated by MDE and MDER. You can think of ME and MER as the HFP equivalents of M and MR, because both generate double-length products of single-length operands. Similarly, MEE and MEER are equivalent to MS and MSR, because both generate single-length products of single-length operands.[215]

To show how ME differs from MEE, suppose we revise Figure 377 on page 600 as shown in Figure 380:

FPR2 Before		Operation	FPR2 After	
////////	////////	LE 2,W	41500000	////////
41500000	////////	ME 2,Y	42230000	00000000

Figure 380. Floating-point registers used for hexadecimal floating-point multiplication

In this case, the product is a *long* hexadecimal floating-point value.

There are no hexadecimal floating-point multiply instructions that use one short and one long operand.

When you use MXD and MXDR, remember these considerations:

- Because the first operand's register pair will contain the extended product, R_1 must be the low-numbered register of a floating-point register pair.
- The fraction part of the low-order half of an extended result is not necessarily normalized (because the 28-digit fraction is normalized in the *high-order* half). Its characteristic is 14 less than the characteristic of the high-order half, except that 128 is added if the low-order characteristic becomes negative. No exponent underflow is indicated if the low-order characteristic becomes negative.

To form the *long* product of the long operand in FPR0 by itself, we write

```
         MDR   0,0
```

To form the *extended* product of the same long operand by itself, we would write

```
         MXDR  0,0
```

and the original contents of FPR2 are replaced by the low-order half of the extended result. To illustrate the use of MXD, suppose we wish to store at **XPrd** the extended product of the two long operands stored at **DA** and **DB**.

215 The ME and MER mnemonics were defined for System/360 and are still valid. MDE and MDER are recommended because they more clearly indicate that multiplying two short operands yields a long product.

```
         LD    0,DA                Load first operand into FPR0
         MXD   0,DB                Form extended product in FPR(0,2)
         STD   0,XPrd              Store high-order half of product
         STD   2,XPrd+8            Store low-order half of product
```

33.7.1. Exponent Overflow and Underflow.

Two error conditions can arise if the result characteristic lies outside the range

$0 \le C_z \le 127$.

If the left inequality is not satisfied (the result characteristic is negative), an *exponent underflow* occurs; if the right inequality is not satisfied (the result characteristic exceeds 127), an *exponent overflow* occurs. These two conditions are sometimes known as exponent or characteristic *spill.* In both cases an interruption condition is recognized, and the quantity left in the result register has the correct sign and fraction, but the characteristic field contains the rightmost seven bits of the true characteristic. This treatment of the characteristic is called “characteristic wraparound”, in the sense that 0 follows 127 for overflows, and 127 follows 0 for underflows.

To illustrate, if we multiply X'70800000' by itself, the product fraction is X'400000'. The result characteristic is

X'70' + X'70' – X'40' = X'A0' (= X'80' + X'20'),

which is the same as X'20' when the rightmost seven bits are retained. The result is then X'20400000', with an overflow interruption.

Similarly, if we multiply X'10800000' by itself, the result is X'60400000'. The result characteristic is found from

X'10' + X'10' – X'40' = –X'20' = –X'80' + X'60'.

By retaining a result which is incorrect only by a fixed power of 16 (16^{128}), you can take steps to continue a calculation and still get a correct final result despite intermediate exponent spills.

Uninterrupted calculation of some results can depend on the order of the operations. Suppose you must multiply the three numbers 16^{50}, 16^{40}, and 16^{-60}. The product of the first two numbers creates an overflow, and multiplying that product by the third number creates an underflow. But, multiplying the first and third numbers and then multiplying by the second generates no interruptions. It helps to know the magnitudes of your operands!

Because exponent underflow means the result has become small enough to be considered negligible in certain circumstances, you can set the HFP Exponent Underflow bit (shown in Table 76 on page 235) in the Program Mask to zero to request that if an exponent underflow occurs, the result should be set to a true zero and no interruption should occur. If the bit is 1, an interruption due to exponent underflow does occur, and the Interruption Code is set to 13 (X'D'). An interruption for exponent overflow cannot be suppressed; the Interruption Code is set to 12 (X'C').

Exercises

33.7.1.(3) Using short operands, modify Figure 378 on page 600 to show how to benefit from a 12-digit product of the operands at **XX** to generate long results at **ZZ**.

33.7.2.(1) What two things can you say that are true for all products generated by ME and MER?

33.7.3.(2) If the ME and MEE instructions rounded their results, could the four high-order bytes be different if they multiply the same operands?

33.7.4.(2) Write a program segment that will compute a table of the cubes of the first 100 integers, and store them as short hexadecimal floating-point numbers starting at **Cubes**.

33.7.5.(2) What result will be in FPR0 after executing these instructions?

```
         LD    0,=X'4111111199999999'
         MEER  0,0
```

33.7.6.(3) Consider the product of any floating-point number with fraction part X'.uvwxyz' and 1.0, which has representation X'41100000'. Show that without a guard digit, the product fraction would be X'.0uvwxy', which after post-normalization becomes X'.uvwxy0'. What does this mean for multiplication by 1?

33.7.7.(1) Show an easy way for the CPU to subtract X'40' from the sum of two characteristics.

33.8. Hexadecimal Floating-Point Division

Table 243 lists the five hexadecimal floating-point divide instructions. The sign of the quotient is set to the XOR of the operand signs. The CC is not changed by a divide instruction.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
7D	DE	RX	Divide (Short)	3D	DER	RR	Divide (Short)
6D	DD	RX	Divide (Long)	2D	DDR	RR	Divide (Long)
				B22D	DXR	RRE	Divide (Extended)

Table 243. Hexadecimal floating-point Divide instructions

Suppose we want to calculate z = w/y. If the divisor fraction F_y is zero, the division is improper and a HFP divide exception is initiated; the resulting program interruption sets the Interruption Code to 15 (X'F'), and the dividend operand in the first-operand register is unchanged. If the dividend fraction F_w is zero, the result is set to zero immediately. Otherwise, both fractions are pre-normalized, and the result characteristic

$C_z = C_w - C_y + X'40' = C_w - C_y + 64$

is adjusted to account for the normalizing shifts. The result fraction $F_z = F_w / F_y$ is then computed. Since we must have a value for F_z that satisfies the inequalities

$16 > F_z \geq 1/16,$

a single corrective right shift may be needed for a proper fractional result; the characteristic is increased by 1 to account for the shift if it is performed, as we saw in Section 32.6.

Unlike fixed-point binary division where the remainder is in the even register and the quotient is in the next higher-numbered odd register, no floating-point remainder is provided by the CPU. We'll see how to calculate a hexadecimal floating-point remainder in Section 33.15.

To give another example of hexadecimal floating-point division, suppose we divide w=3 by y=5. Then,

F_w = X'.3', F_y = X'.5',
$C_w = C_y$ = X'41'
C_z = X'41'−X'41'+X'40' = X'40'.

We know that the result fraction in decimal is 0.6; to convert this to hexadecimal we use the method described in Section 31.2. on page 553:

```
   0.6
 ×  16
   9.6
```

so the first hexadecimal fraction digit is 9. But the next decimal digit is 6 again, so that 0.6_{10} = X'.9999999...' and F_z = X'.999999'.

In short floating-point form, the quotient z is X'40999999'. This result is not rounded; the rounded result would have been X'4099999A'. Thus,

```
         LE    0,=E'3'             Load FPR0 with 3  (X'41300000')
         DE    0,=E'5'             Divide by 5       (X'41500000')
```

leaves the unrounded quotient X'40999999' in FPR0.

The hexadecimal floating-point divide instructions DE, DER, DD, and DDR are similar to the four multiply instructions, except that the divisor, dividend, and quotient fractions are all 6 digits long for short operand division, 14 digits for long operand division, and 28 digits long for extended division.

No attempt is made to provide an "inverse" to the short multiply instructions by dividing a short divisor into a long dividend. The low-order part of the register is ignored in short HFP division, and is *not* replaced by a remainder.

If you're not dividing by zero, the quotient is computed; if a corrective right shift of the result fraction is needed, the extra digit at the right end is lost and the quotient then replaces the dividend. Characteristic wraparound also applies to division: if we divide X'60200000' by X'10400000' the quotient is X'10800000' with an exponent overflow indicated, and if we divide X'16600000' by X'79200000', the result is X'5D300000' with a (maskable) exponent underflow condition indicated.

Figure 381 illustrates floating-point divide instructions. Suppose we again have two arrays at **XX** and **YY** of ten short hexadecimal floating-point operands X(i) and Y(i), and wish to store $Z(i)=|X(i)|/Y(i)^2$ in the array at **ZZ**.

```
         XR    7,7                 Set index in GR7 to zero
         LA    2,10                Initialize counter in FR2 to 10
Loop     LE    2,XX(7)             Load X(i) in FPR2
         LPER  4,2                 Place absolute value in FPR4
         LE    6,YY(7)             Get Y(i)
         MER   6,6                 Square it in FPR6
         DER   4,6                 Form quotient in FPR4
         STE   4,ZZ(7)             Store Z(i)
         LA    7,4(,7)             Increment index
         JCT   2,Loop              Count and loop
```

Figure 381. Example of hexadecimal floating-point divide instructions

It was unnecessary to use FPR4 as an intermediate register. Figure 382 shows another way to do the same calculation. (It might run slightly slower because more divide instructions are used and more memory accesses are required for the operands.)

```
         XR    1,1                 Initialize index
         LA    2,10                And counter
Loop     LE    0,XX(1)             X(i) in FPR0
         LPER  0,0                 Abs(X(i)) in FPR0
         DE    0,YY(1)             Abs(X(i))/Y(i)
         DE    0,YY(1)             Abs(X(i))/Y(i)/Y(i)
         STE   0,ZZ(1)             Store Z(i)
         LA    1,4(,1)             Increment index
         JCT   2,Loop              Count down and branch
```

Figure 382. Example of hexadecimal floating-point divide instructions

33.8.1. The Halve Instructions (*)

The two instructions in Table 244 divide an operand by 2 by doing a single binary shift, saving the cost of a more expensive multiplication or division instruction. The first operand is replaced by a number whose value is one-half that of the second operand. The Condition Code is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
34	HER	RR	Halve (Short)	24	HDR	RR	Halve (Long)

Table 244. Hexadecimal floating-point Halve instructions

The fraction part of the second operand is shifted right by one *bit* position. The bit shifted off the right end moves into the guard digit, which was initialized to zero. If the fraction is still normalized, the instruction is complete; otherwise a normalizing (hexadecimal) left shift is done, and the characteristic is decreased by 1 for each shift. The result is not rounded.

For example:

```
LE    0,=E'10'          c(FPR0) = X'41A00000'
HER   0,0               Result  = X'41500000' (no shift)

LE    0,=E'1.0'         c(FPR0) = X'41100000'
HER   0,0               Result  = X'40800000' (normalized)
```

Figure 383. Example of a hexadecimal floating-point halve instruction

The only possible exception is an exponent underflow, as illustrated in Figure 384.

```
LE    0,=EH'(Min)'      c(FPR0) = X'00100000'
HER   0,0               Result  = X'7F800000' with underflow
```

Figure 384. Hexadecimal halve instruction causing underflow

For short operands (using the HER instruction), the low-order part of the register is unchanged, and does not contain the lost bit when no post-normalization is performed. If the fraction part of the second operand is zero, a true zero replaces the first operand.

Exercises

33.8.1.(1) In calculating the quotient 3/5, the result is given as X'40999999', and not the rounded value X'4099999A'. How do we know what the rounded answer is?

33.8.2.(2)+ What is the result of executing

```
HER   0,0
```

if FPR0 contains each of these operands:

1. X'4013579B'
2. X'40013579'

33.8.3.(2)+ What is the result of executing

```
HDR   0,0
```

if FPR0 contains each of these operands:

1. X'42FB2690 5A66B73D'
2. X'C5774302 D4FB018F'

33.8.4.(2) What result will be in FPR4 after executing these instructions?

```
LE    0,=X'04000002'
HER   4,0
```

33.8.5.(1) What instructions involving the general registers might be thought of as the fixed-point analogs of the floating-point Halve instructions HER and HDR?

33.8.6.(3) Write instructions to implement a *rounded* hexadecimal floating-point “halve” operation.

33.9. Hexadecimal Floating-Point Addition and Subtraction

There are two major groups of hexadecimal floating-point add and subtract instructions: those that normalize their results, and those that do not. Both groups are listed in Table 245.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
7A	AE	RX	Add (Short)	3A	AER	RR	Add (Short)
7B	SE	RX	Subtract (Short)	3B	SER	RR	Subtract (Short)
6A	AD	RX	Add (Long)	2A	ADR	RR	Add (Long)
6B	SD	RX	Subtract (Long)	2B	SDR	RR	Subtract (Long)
36	AXR	RR	Add (Extended)	37	SXR	RR	Subtract (Extended)
7E	AU	RX	Add (Short) Unnormalized	3E	AUR	RR	Add (Short) Unnormalized
7F	SU	RX	Subtract (Short) Unnormalized	3F	SUR	RR	Subtract (Short) Unnormalized
6E	AW	RX	Add (Long) Unnormalized	2E	AWR	RR	Add (Long) Unnormalized
6F	SW	RX	Subtract (Long) Unnormalized	2F	SWR	RR	Subtract (Long) Unnormalized

Table 245. Hexadecimal floating-point Add/Subtract instructions

Four distinctive properties of the hexadecimal floating-point add and subtract instructions are:

1. No initial tests are made for zero operands.
2. Neither fraction is pre-normalized.
3. The Condition Code is always set.
4. The operation code specifies whether or not the final result will be normalized.

The first step is to compare the characteristics of the operands. If the difference is such that the fraction part of the operand with the smaller characteristic would have to be shifted to the right by too many places so that no significant digits would be left, the result is the operand with the larger characteristic. That is, the smaller operand is ignored in short arithmetic if the magnitude of the characteristic difference is greater than 6, in long arithmetic if it exceeds 14, and in extended arithmetic if it exceeds 28, because there could be no useful digit to add. In such cases, no shifted digit could appear in the guard digit position.

For example, suppose we add the normalized short operands X'58123456' and X'52987654'. The characteristic difference X'58'−X'52' = 6 means that the second fraction must be shifted right by six digit positions, giving a fraction with six leading zero digits. Since the intermediate arithmetic is performed with a precision of only seven digits, the second operand cannot possibly contribute to the final result, so it is ignored. (Remember that the guard digit is initialized to zero!) To visualize this process:

```
 58123456(0)          First operand, guard digit 0
+52987654             Second operand before shifts

 58123456(0)          First operand, guard digit 0
+58000000(9)          Second operand after shifts
 58123456             Sum
```

Figure 385. Example of hexadecimal floating-point addition

Because the sum need not be normalized, the guard digit is lost.

Subtraction can be different; consider the same two operands:

```
 58123456(0)              First operand, guard digit 0
-52987654                 Second operand before shifts

 58123456(0)              First operand, guard digit 0
-58000000(9)              Second operand after shifts
 58123455                 Difference
```

In this case, the digit shifted into the guard digit position causes a "borrow" from the low-order digit of the first operand. Thus, if the operation is a subtraction, an exponent difference equal to the number of digits in the significand does not necessarily mean that the result will be the larger operand.

For addition, if the characteristic difference is *equal* to 6, 14, or 28, and if the operand with the larger characteristic is unnormalized, a digit from the operand with the smaller characteristic may appear in the result. (See Exercise 33.9.14.)

If the registers were longer, the second operand could contribute to the sum, so short and long arithmetic might yield different results from the same data. (See Exercise 33.9.1.)

Adding (or subtracting) two numbers that lead to a zero fraction produces different results, depending on the setting of the Program Mask. If the intermediate result fraction including the guard digit is zero, a HFP *significance exception* may be recognized, meaning there are no significant digits in the result, only the correct sign and characteristic.

As with exponent underflow, you can zero the HFP Significance bit in the Program Mask (shown in Table 76 on page 235) to request that the CPU ignore the exception condition and set the result to a true zero.[216] If the interruption does occur, the result register will contain a pseudo-zero with the correct characteristic, and the Interruption Code will be set to 14 (X'E').

For example:

```
LE   2,=X'456789AB'
AE   2,=X'C56789AB'
```

If the Program Mask bit is zero, c(FPR2)=X'00000000'; but if the Program Mask bit is one, c(FPR2)=X'45000000' when the program interruption occurs.

The difference between a true zero is and a pseudo-zero is rarely important, so most applications set the HFP-significance mask bit to zero so they don't have to worry about interruptions for calculated zero results.

Suppose we must evaluate the expression

```
Z = T * (A + B) + 2C / (A + B)
```

where all operands are long and are found at memory locations with the same names.

```
LD    6,A              Form sum first
AD    6,B              (A+B) in FPR6
HDR   4,6              Save (A+B)/2 in FPR4
MD    6,T              Multiply sum by T, giving T*(A+B)
LD    2,C              Load C in FPR2
DDR   2,4              Divide by c(FPR4) giving 2C/(A+B)
ADR   2,6              Form Z in FPR2
STD   2,Z              Store the long result at Z
```

Figure 386. Evaluating a hexadecimal floating-point expression

[216] In practice, the Significance exception is rarely enabled because it would occur even when a calculated result *should* be zero. The Program Mask bit is almost always set to zero. (You can change the bits in the Program Mask with the SPM instruction, as described in Section 16.2.)

Suppose we must form the “inner product” of two linear arrays of twenty short-precision elements each, by evaluating the sum

```
Z = X(1)*Y(1) + X(2)*Y(2) + ... + X(19)*Y(19) + X(20)*Y(20),
```

where the arrays of the X(i) and Y(i) values are stored starting at **XX** and **YY** respectively. The value of Z will be accumulated in long precision, and stored in short precision.

```
TLen     EQU   20                  Table length is 20
         LZDR  2                   Initialize sum in FPR2 to zero
         XR    7,7                 Initialize index to zero
         LA    2,4                 Increment = 4
         LA    3,4*(TLen-1)        Comparand = 76
Loop     LE    0,XX(7)             C(FPR0) = X(i)
         ME    0,YY(7)             C(FPR0) = X(i)*Y(i) (long result)
         ADR   2,0                 Add long product to retain accuracy
         JXLE  7,2,Loop            Increment index and branch
         STE   2,Z                 Store short floating-point result
```

Figure 387. Evaluating a hexadecimal floating-point inner product

As a second example, suppose we want to evaluate the polynomial

$$Z = A(10)*Y^{10} + A(9)*Y^{9} + A(8)*Y^{8} + ... + A(1)*Y^{1} + A(0),$$

where the coefficients A(k) are stored in an array starting at **AA** in the order A(0), A(1), ..., A(10). We will do the evaluation by writing the polynomial in “nested” form (sometimes known as Horner's Rule):

```
Z = ((...((A(10) * Y) + A(9)) * Y + ... + A(1)) * Y + A(0))
```

Figure 388 shows an example using this method.

```
LE       EQU   10                  Highest coefficient number
         LHI   3,-4                JXH increment and comparand in GR3
         LA    2,4*(LE-1)          Initial index points to A(9)
         LE    6,AA+4*LE           Initial polynomial term is A(10)
         LE    4,Y                 Hold Y in FPR4, less memory access
Next     MER   6,4                 Multiply current sum by Y
         AE    6,AA(2)             Add next coefficient
         JXH   2,3,Next            Decrease index by 4 and loop
         STE   6,Z                 Store short result
```

Figure 388. Evaluating a polynomial with hexadecimal floating-point arithmetic

For short polynomials it is simpler and faster to do the calculation without looping. If we want only the quadratic portion (up to the second power in Y) of the same polynomial, we could use the instructions in Figure 389:

```
         LE    0,AA+8              Get A(2)
         ME    0,Y                 *Y
         AE    0,AA+4              +A(1)
         ME    0,Y                 *Y
         AE    0,AA                +A(0)
         STE   0,Z                 Store at Z
```

Figure 389. Evaluating a quadratic polynomial

and no loop-control “housekeeping” instructions are needed.

33.9.1. Unnormalized Addition and Subtraction

Unnormalized addition and subtraction follow the same rules as the normalizing operations, except that the result is not post-normalized. This leads to these conditions:

- The guard digit is ignored (it participates only if a post-normalization shift is needed to deliver a normalized result).
- If the sum creates a carry out of the high-order digit, a corrective right shift is needed, so an exponent overflow is possible. For example:

```
         LE    0,=X'7FFE7654'      Large number in FPR0
         AU    0,=X'7EFEDCBA'      Add another almost as large
```

  generates an exponent overflow interruption, and the result in FPR0 is X'0010E641'.
- If there is no overflow, the characteristic of the result is that of the operand with the larger characteristic (the fraction of the operand with the smaller characteristic will have been shifted right).
- Because there is no normalizing left shift, no exponent underflow is possible.
- If the result fraction is zero, a HFP lost-significance exception is possible; if suppressed, the result is set to +0. For example:

```
         LE    0,=X'76543210'      Initialize FPR0
         SUR   0,0                 Unnormalized subtraction
```

  generates a significance-exception interruption, and the result in FPR0 is X'76000000'. If the Program Mask bit is zero there is no interruption, and c(FPR0)=X'00000000'.

The test for a zero intermediate fraction *includes* the guard digit for normalized addition and subtraction, and *excludes* the guard digit for unnormalized addition and subtraction, because the guard digit cannot appear in the final result. (See Exercise 33.9.9.)

33.9.2. Older Uses of Unnormalized Addition (*)

Some history

This section describes instructions that see relatively little use on modern CPUs.

Before the instructions described in Section 33.14 on page 620 were available, unnormalized addition was needed to convert numbers between binary integers and HFP values. If we had to convert 12345 from binary to hexadecimal floating-point we might have used the instructions in Figure 390.

```
         L     1,=F'12345'        Binary integer to be floated
         ST    1,Const+4          Store in right half of a constant
         LD    0,=D'0'            Clear FPR0
         AD    0,Const            Add the constant
         - - -
Const    DC    0D,X'4E',7X'0'     Unnormalized constant
```

Figure 390. Converting a binary integer to hexadecimal floating-point

After the ST instruction, c(Const)=X'4E00000000003039'. Because its exponent is 14, the radix point is at the right end of the constant; after executing the AD instruction to normalize the result, c(FPR0)=X'4430390000000000'.

Converting in the other direction used a similar technique shown in Figure 391 on page 610, this time with an unnormalized addition:

```
LD    0,=D'12345'        Value to be converted to binary
AW    0,=X'4E00000000000000'  Force unnormalization
STD   0,DTemp            Store the result
L     1,DTemp+4          Binary integer now in GR1
```

Figure 391. Converting a hexadecimal floating-point number to a binary integer

After the AW instruction is executed, c(FPR0)=X'4400000000003039'. The L instruction finally loads X'00003039'=12345 into GR1.

Newer instructions provide improved ways of converting between HFP and binary, as we'll see in Section 33.14.

Exercises

33.9.1.(2) In Figure 385 on page 606, we considered adding the short operands X'58123456' and X'52987654'. If these were the *long* operands X'5812345600000000' and X'5298765400000000', what would be their sum?

33.9.2.(1) Suppose you execute these instructions:

```
LD    2,=D'1.0'
LD    6,=D'2.5'
AXR   2,6
```

What will be in FPR2?

33.9.3.(0) There is an engineer's pun among the mnemonics in Table 245 on page 606. Can you find it?

33.9.4.(2)+ Given the following short hexadecimal floating-point operands:

```
A = X'41200000'
B = X'C0138762'
C = X'420A0000'
D = X'6B200000'
E = X'45000000'
```

Show (in hexadecimal) the appropriate contents of FPR0 and the setting of the Condition Code after executing each of the following instruction sequences.

```
1. LE   0,E        2. LE   0,D         3.  LE   6,E        4. LE   0,C
   SE   0,B           MER  0,0             LTER 0,6            DE   0,D
```

33.9.5.(3)+ Given the following hexadecimal floating-point quantities:

```
A = X'C1200000'
B = X'4310000F'
C = X'03780000'
D = X'C400E000'
E = X'408279ED'
```

Show (in hexadecimal) the contents of FPR0 and the Condition Code setting after executing each of the following instruction sequences:

```
1.  LE  0,A        2.  LE  0,E         3.  LE  4,B         4.  LE  0,B
    AE  0,B            AER 0,0             HER 0,4             DE  0,C

5.  LE  0,C        6.  LE  0,A
    MER 0,0            AE  0,E
```

33.9.6.(3)+ Using the same quantities as in Exercise 33.9.5, and these additional hexadecimal floating-point quantities:

```
W = X'04100000'
X = X'0B012335'
Y = X'4600ABCD'
```

show (in hexadecimal) the contents of FPR0 and the Condition Code setting after executing each of the following instruction sequences:

```
1. LE  0,A        2.  LE   0,D       3.  SER 0,0         4.  LE  0,X
   AU  0,C            AU   0,B                               SU  0,X

5. LE  0,W        6.  LE   0,Y
   AU  0,X            AUR  0,0
```

33.9.7.(2)+ Sometimes programs use data having the wrong type, as in this example. What result will be in FPR0 after executing these instructions?

```
LE    0,=F'-3'          Binary integer?
AER   0,0               Added to itself as floating-point??
```

33.9.8.(1) Show that if an exponent overflow occurs in executing an unnormalized add or subtract instruction, the characteristic of the result must be zero.

33.9.9.(2)+ Suppose `X'40FFFFFF'` is subtracted from `X'41100000'` using (1) SE and (2) SU instructions. What will be the result in each case?

33.9.10.(1) In Figure 388 on page 608, what conflicts might arise between the uses of LE as a symbol and as a mnemonic?

33.9.11.(2) An array of short hexadecimal floating-point numbers is stored starting at **Data**, and the number of elements in the array is in the fullword integer at **NItems**. Write instructions to compute the average of the numbers and store the result at **Average**.

33.9.12.(2) Suppose you want to add 1000000 and 0.000001 using hexadecimal floating-point arithmetic. What limitations should be imposed on the registers and operands if the result is to be computed correctly?

33.9.13.(2)+ Suppose you add `X'58023456'` to `X'52987654'`. The difference in the characteristics is 6; what is the result of the addition?

33.9.14.(2) Show the result in FPR0 after executing each of the following five pairs of instructions:

```
(1)       LE    0,=X'41100005'
          AE    0,=X'40100005'

(2)       LE    0,=X'41100005'
          SE    0,=X'40800005'

(3)       LE    0,=X'41100000'
          SE    0,=X'38100000'

(4)       LE    0,=X'41100000'
          SE    0,=X'3A100000'

(5)       LE    0,=X'40123456'
          AE    0,=X'40100000'
```

A Suggestion

The next two subsections (33.10 and 33.11) explore some details of implementing floating-point addition and subtraction. If you're not interested, they can be skipped with no loss in continuity.

33.10. Adding Operands of Like Sign (*)

Adding operands of like sign (or *true* addition) is straightforward; the sign of the result is known. If the original operands are normalized, the result fraction cannot need any post-normalizing left shifts because no leading zero digits can be generated, and at most one corrective right shift (with an accompanying increment of the characteristic) can be needed. Thus, true addition is the only case in floating-point addition where exponent overflow can occur. The guard digit is significant in like-sign addition only if unnormalized operands are used, *and* if the instruction also requires that the result be normalized. A few examples of adding short operands will illustrate these points.

Example 1: `X'40111111' + X'40222222'`

Because the exponents are equal, the fractions are added directly, with result `X'40333333'`. The CC is set to 2 to indicate a positive result.

Example 2: `X'40123456' + X'42456789'`

After shifting the fraction part of the smaller first operand right by two places, the intermediate result is `X'424579BD(5)'`, where the guard digit is parenthesized. The CC is again set to 2.

Example 3: `X'42FEDCBA' + X'41456789'`

After shifting the second fraction to the right by one place, the intermediate sum is `X'1.033332(9)'` with a carry bit (so seven digits *plus* a guard digit are shown). After a corrective right shift, the result is `X'43103333'`; *both* the guard digit and the low-order digit of the intermediate sum are lost, and no rounding is done. The CC is set to 2.

Example 4: `X'FF801020' + X'FF934567'`

This case illustrates an exponent overflow: the result fraction is `X'1.145587(0)'`, which after right-shifting leads to characteristic `X'80'` and fraction `X'.114558(7)'`. The guard digit is lost, and because of characteristic wraparound the result becomes `X'80114558'`, and an exponent overflow interruption is initiated. The CC is set to 1 indicating a negative result.

Exponent overflow during addition should be an infrequent occurrence, because the operands involved must be quite large.

Example 5: `X'43000123' + X'416789AB'`

In this case the intermediate sum fraction `X'.0068AC(A)'` is unnormalized; if the result is normalized, we obtain `X'4168ACA0'`, and the guard digit is retained.

Example 6: `X'02000012' + X'00023456'`

After right-shifting the second fraction by two places, the intermediate sum is `X'02000246(5)'`. On normalization, this gives `X'7F246500'`, with an accompanying exponent underflow exception condition.

If the Exponent Underflow mask bit is zero, no interruption occurs and the CC and the result are set to zero. Otherwise, the result register is unchanged, the CC is set to 2 (the result is positive), and an interruption occurs.

Exponent underflow can occur when adding numbers of like sign only if both operands are unnormalized.

33.11. Adding Operands of Unlike Sign (*)

Adding operands of unlike sign (or *complement* addition) is more complicated.

1. No exponent overflow can be generated, because the difference of two operands is never greater in magnitude than the larger of the original two.
2. Leading zeros can be generated in the fraction part of the intermediate result so that exponent underflows can occur even with normalized operands.

Suppose we add −5 (X'C1500000') and +121 (where 121 = X'79' = X'.79'×16^2 (X'42790000'). After shifting the first fraction to the right by one digit position and subtracting it from the second fraction, the intermediate result is X'42740000(0)'. Since the result is now normalized and no characteristic spill was generated, the result is X'42740000' and the CC is set to 2 indicating a positive result. If we use the same operands with inverted signs, the same process generates X'C2740000', with CC 1 indicating a negative result.

33.11.1. Hexadecimal Floating-Point Complement Addition (*)

It's easy in some cases to see what the result should be, by inspecting the operands: we observe which is the larger, subtract the smaller, and use the sign of the operand with the larger magnitude.

But the CPU cannot compare the operands to see which has the larger magnitude without doing a subtraction! It would have to duplicate much of its effort if a comparison indicated that the subtraction should actually have been done in the opposite order. Thus, a technique similar to that for fixed-point arithmetic is used, where subtraction was performed by adding the two's complement of the second operand to the first.

We can't do the same with the fraction parts of the floating-point operands, because the sign bit is not an extension of the fraction: that is, the fractions are represented in a *sign-magnitude* representation.

Though System z may not do complement addition exactly this way, we will describe a method, using the short-arithmetic addition of −5 and +121.

1. Given operands X'C1500000' and X'42790000', determine whether one of them has a characteristic larger than the other; call it the first operand. (Thus, X'42790000' is the first operand.) If the characteristics are equal, simply choose one of the operands as the first.
2. As in true addition, if the characteristic difference exceeds 6 for short, 14 for long, or 28 for extended operands, the intermediate result is the first operand.
3. As in true addition, shift the fraction part of the second operand to the right by a number of places equal to the magnitude of the characteristic difference. In our example, the second operand fraction is shifted once to produce X'.050000(0)', with the seventh digit being the guard digit.
4. The sign of the intermediate result is assumed to be the same as the sign of the first operand ("+" in this example).
5. The two's complement of the second fraction from step 3, including its guard digit, is added to the first fraction with its guard digit, with carries off the left being ignored. Thus, we add X'.790000(0)' + X'.FB0000(0)', giving X'(1).740000(0)'. The carry bit has been explicitly indicated, even though it doesn't appear in the result.

   The two's complement of the second fraction is calculated this way:

   ```
     .500000      original value
     .050000(0)   (step 3) shift right 1 digit; last digit becomes guard digit
     .FAFFFF(F)   ones' complement
   + .000000(1)   add 1 to form two's complement
     .FB0000(0)   two's complement of fraction from step 3
   ```
6. Depending on the presence or absence of a carry, one of the following is performed:
   a. if there *is* a carry, it is ignored and the intermediate result consists of the first operand sign, the first operand characteristic, and the intermediate fraction to whatever precision it was calculated.
   b. if there is *no* carry, the intermediate result fraction is said to be in *complement* form. It is then recomplemented, and the sign bit is inverted.

At this point the zero-fraction test for a significance exception is made; the guard digit does not participate if an unnormalized result is to be generated. The intermediate result may be normalized if required by the instruction.

Here are some further examples of adding operands, now with unlike signs.

Example 7: `X'C2777444' + X'40121314'`

The first operand is `X'C2777444'`; after shifting and then complementing the second operand fraction, we do the following addition (where we show the parts of the intermediate result as sign, characteristic, and fraction):

```
Sign Char        Fraction
 -    42         .777444(0)  first operand with guard digit
                 .FFEDEC(F)  complemented 2nd operand fraction
 -    42      (1).776230(F)  carry occurs
```

Thus the result is in true form, and we have the intermediate result `X'C2776230(F)'`. The lowest-order digit of the second operand was lost in the shifting operation.

Example 8: `X'C2000444' + X'40121314'`

As before, we arrive at

```
 -   42          .000444(0)  first operand
                 .FFEDEC(F)  complemented 2nd operand fraction
 -   42          .FFF230(F)  no carry occurs
```

The result is in complement form, as indicated by the absence of a carry. After recomplementation and sign inversion, we have

```
 +   42          .000DCF(1)
```

giving the intermediate result `X'42000DCF(1)'`. The guard digit will become significant if the final result is normalized, when the result would be `X'3FDCF100'`.

Example 9: `X'42111111' + X'C1111111'`

The intermediate calculation would yield

```
 +   42          .111111(0)
                 .FEEEEE(F)
 +   42       (1).0FFFFF(F)
```

with a carry; thus we have the intermediate result `X'420FFFFF(F)'`, and normalized result `X'41FFFFFF'`.

Example 10: `X'40123456' + X'C0123457'`

The intermediate result would be found from

```
 +   40          .123456(0)
                 .EDCBA9(0)
 +   40          .FFFFFF(0)
```

with no carry; after recomplementation and sign inversion the intermediate result would be `X'C0000001(0)'`, as we would have guessed from inspecting the operands.

These last two examples illustrate an important consideration: if the original operands are normalized, a single guard digit provides adequate protection against unexpected loss of precision. This is because the difference of two normalized fractions with unequal characteristics can introduce at most *one* leading zero digit in the intermediate result fraction (except in cases of severe cancellation). Example 8 above illustrates that a single guard digit may be insufficient if unnormalized operands are allowed. If the operands have the same characteristic (as in Example 10), the possible loss of accuracy can be predicted but not prevented, so in this situation you are entitled only to the digits left in the intermediate result and no more.

33.11.2. Implementing Hexadecimal Floating-Point Complement Addition (*)

Complement addition can be simplified by forming the *ones'* complement of the second fraction, rather than the two's complement. Then, if a carry occurs after the fractions are added, it is "carried around" to the *low-order* end of the intermediate sum and added there. This gives the effect of having added the two's complement of the second operand initially. If there is no carry, the recomplementation is also a ones' complement, which "retrieves" the low-order one-bit that would have been added if the two's complement had been formed originally; that is, we have effectively subtracted and then added back the low-order bit.

A reason for using this technique is that the formation of the two's complement of a number is an *arithmetic* operation requiring an addition (and therefore, time and hardware for the propagation of carries), while ones' complementation requires only the simple and inexpensive logical operation of inverting all the bits of an operand. (You might review the similar discussion at "How Additions and Subtractions are Actually Performed" on page 34.)

Using this revision to the above addition scheme, Examples 9 and 10 above would appear as shown in Examples 11 and 12 below.

Example 11: `X'42111111' + X'C1111111'`

The intermediate calculation gives

```
+   42       .111111(0)  first operand
             .FEEEEE(E)  ones' complement of 2nd fraction
          (1).0FFFFF(E)  carry occurs, so...
            +        1   add back at low-order end
+   42       .0FFFFF(F)  intermediate result
```

Example 12: `X'40123456' + X'C0123457'`

Remember that a guard digit is part of the second operand fraction, even though it is initially zero. The intermediate calculation is

```
+   40       .123456(0)  first operand
             .EDCBA8(F)  ones' complement of 2nd fraction
+   40       .FFFFFE(F)
```

The intermediate result fraction is in complement form; recomplementation (now using the *ones'* complement) and sign inversion give `X'C0000001(0)'` as before.

Exercises

33.11.1.(3) In Example 8 above, what result would be generated if the `X'C2000444'` operand was normalized?

33.11.2.(3)+ Given the hexadecimal floating-point quantities you defined in Exercise 33.1.5, compute in decimal and in then in short hexadecimal floating-point forms the following values:

(1) D+C (2) A+E (3) B+E (4) E−F (5) D×B (6) C×C (7) D/B

33.11.3.(2) In Example 10 of Section 33.11.1, what is the normalized result?

33.12. Hexadecimal Floating-Point Comparison

Table 246 lists the hexadecimal floating-point comparison instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
79	CE	RX	Compare (Short)	39	CER	RR	Compare (Short)
69	CD	RX	Compare (Long)	29	CDR	RR	Compare (Long)
				B369	CXR	RRE	Compare (Extended)

Table 246. Hexadecimal floating-point Compare instructions

Comparing two floating-point operands requires an *internal* subtraction which follows all the above rules. As with other compare instructions, no interruption conditions are recognized and the intermediate result (*including* the guard digit) is examined to determine the CC setting. The operands are considered to be equal if the intermediate difference (including the guard digit) is zero.

The Condition Code settings are shown in this table:

CC	Meaning
0	Operands are equal
1	First operand is low
2	First operand is high

Table 247. CC settings for hexadecimal floating-point comparison

Be careful: if you compare unnormalized operands, two unequal operands may compare equal. For example, if we compare the short HFP number X'40000001' to any number of the form X'3B10wxyz', the CC setting is zero (indicating equality) for any values of the digits "wxyz". (See Exercise 33.12.1.)

Exercises

33.12.1.(2)+ Perform the intermediate steps of comparing the two short operands X'40000001' and X'3B10wxyz', where wxyz are arbitrary hex digits, and prove that they will be found equal.

33.12.2.(2) Given the quantities A, B, C, D, and E shown in Exercise 33.9.5, show the CC setting after executing each of these instruction sequences:

```
1. LE   0,D        2. LE   0,C          3.  LE   6,E        4. LE   0,B
   CE   0,A           CE   0,D              CE   6,E           CE   0,E
```

33.12.3.(3) Given three arrays of 10 short hexadecimal floating-point values stored beginning at **SideA**, **SideB**, and **SideC** respectively, write an instruction sequence to set the byte at offset K in the ten-byte string starting at **Triangle** to one if the values of the K-th floating-point numbers could form the sides of a plane triangle, and to zero if not. (Remember that a triangle has the property that the sum of any two sides is greater than the third.)

33.12.4.(3) Given two arrays of twenty short HFP numbers X(i) and Y(i) stored at **XX** and **YY** respectively, write an instruction sequence that will store in the array starting at **Ratio** the quantity (X(i) / Y(i)) if $|X(i)/Y(i)| < 10^{40}$, or 10^{40} otherwise.

Your program should be written so that no exponent overflows or underflows can be generated.

33.12.5.(3)+ Under what circumstances can two hexadecimal floating-point values with different exponents compare equal?

33.13. Rounding and Lengthening Instructions

Two sets of length-changing hexadecimal floating-point instructions let you

- round a longer operand to a shorter, and
- extend a shorter operand to a longer.

33.13.1. Rounding Instructions

Though hexadecimal floating-point arithmetic is not rounded, the instructions in Table 248 round a longer operand to a shorter.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
35	LRER, LEDR	RR	Load Rounded (Long to Short)				
B366	LEXR	RRE	Load Rounded (Extended to Short)	25	LRDR, LDXR	RR	Load Rounded (Extended to Long)

Table 248. Hexadecimal floating-point Round instructions

The LRDR and LXDR instructions round an extended operand to long, and LRER and LEDR round a long operand to short, as follows.

1. Examine the fraction bit of the next longer floating-point format that immediately follows the low-order fraction bit of the shorter operand. For LRER, LEDR, or LEXR this will be bit 32 of FPR(R_2); and for LRDR or LDXR, it will be bit 8 of FPR(R_2+2).
2. If the examined bit is zero, the high-order part of the second operand is placed in the first operand register.
3. If the examined bit is one, a low-order 1-bit is added internally to the low-order bit of the first-operand length of the second operand, and the resulting floating-point sum is placed in the first operand register.
4. If, in adding the low-order 1-bit, a carry occurs out of the high-order digit of the fraction, the fraction is shifted right one hex digit, and the characteristic is increased by 1. If this process causes the characteristic to exceed 127, an exponent overflow interruption occurs, and the IC is set to 12 (X'C').
5. No post-normalization is done, except for a possible right shift of one digit if there is a carry out of the high-order digit position, and the characteristic is incremented.
6. The Condition Code is not changed by these instructions.

For example, to round a long operand in FPR2 to short and store it at **Rounded** we can use the instructions in Figure 392.

```
         LRER  0,2                  Long operand in FPR2, short in FPR0
         STE   0,Rounded            Store short rounded result
```

Figure 392. Rounding a long hexadecimal floating-point number to short

To form the extended product of two long operands at **DA** and **DB** and then round the result to long format, you can use an instruction sequence like this:

```
         LD    0,DA                 Load first operand into FPR0
         MXD   0,DB                 Form extended product in FPR(0,2)
         LRDR  0,0                  Round extended to long in FPR0
```

Suppose we must store at **ZZ** the rounded inner product of two linear arrays of twenty long-precision elements that start at **XX** and **YY**. (Compare Figure 387 on page 608.)

```
TL       Equ   20                   Length of arrays
         LZXR  4                    Accumulate extended sum in FPR(4,6)
         LA    0,L'XX               JXLE Increment in GR0
         LA    1,L'XX*(TL-1)        JXLE Comparand in GR1
         SR    2,2                  JXLE Index in GR2
Loop     LD    0,XX(2)              Fetch operand XX(i)
         MXD   0,YY(2)              Form extended product XX(i)*YY(i)
         AXR   4,0                  Accumulate extended sum
         JXLE  2,0,Loop             Increment index and loop
         LRDR  2,4                  Round the result sum into FPR2
         STD   2,ZZ                 Store final result
         - - -
XX       DS    (TL)D                Space for first array
YY       DS    (TL)D                Space for second array
ZZ       DS    D                    Rounded inner product stored here
```

Figure 393. Rounded inner product of long HFP numbers

To retain precision, products of the long elements are generated and accumulated as an extended-precision sum that is rounded to the final long result. Because there are no instructions to test individual bits in the floating-point registers, you could do rounding “manually” in other ways that illustrate the rounding method used by the CPU.

Suppose the LRER instruction did not exist. We could write instruction sequences to round the long floating-point number in FPR2 in several ways:

1. A direct method stores the long operand and tests the bit. If it is one, form an unnormalized quantity of the same sign and characteristic with only a low-order 1-bit in the fraction, and add it to do the rounding. Suppose the result is to be stored at **Round1**.

```
         STD   2,Temp            Store long operand
         TM    Temp+4,X'80'      Test high-order bit of right half
         JZ    Done              If zero, no rounding
         MVC   One(1),Temp       Move sign and characteristic
         AE    2,One             Add roundoff bit
Done     STE   2,Round1          Store rounded result
         - - -
Temp     DS    D                 Workspace
One      DC    F'1'              Low-order 1-bit for rounding
Round1   DS    E                 Rounded result
```

Figure 394. Manually rounding long to short (1)

Suppose FPR2 initially contains `X'ccdd dddr xeee eeee'`, where `cc` is the characteristic, `r` is the digit that might be rounded, and the high-order bit of `x` is tested by the TM instruction. If the bit is zero, nothing else need be done. If it is 1, the characteristic is moved to the high-order byte of the "constant" named **One** (poor programming technique!), forming `cc00 0001`, which is then added as a short hexadecimal floating-point value to FPR2, rounding the long value to short.

2. Another instruction sequence uses the fact that if the high-order bit of the lower half of the long operand is a 1-bit, then adding that bit to itself will produce a carry into the low-order bit position of the short half of the register.

```
         STD   2,T               Store long operand
         XC    T+1(3),T+1        Set 1st 3 fraction bytes to 0
         AD    2,T               Add back to round off
         STE   2,Round2          Store rounded short operand
         - - -
T        DS    D
Round2   DS    E
```

Figure 395. Manually rounding long to short (2)

3. This instruction sequence uses only the floating-point registers, and no intermediate storage:

```
         LDR   6,2               Move long argument to FPR6
         LZDR  4                 Clear all of FPR4
         LER   4,2               Move short half of argument to FPR4
         SWR   6,4               Subtract left part of fraction
         ADR   2,6               Add low-order half to round
         STE   2,Round3          Store rounded result
         - - -
Round3   DS    E
Round1   DS    E                 Rounded result
```

Figure 396. Manually rounding long to short (3)

Remember that SWR is an unnormalized subtract instruction (see Table 245 on page 606).

These examples only show how rounding works; it's far simpler to use the CPU's rounding instructions, not these.

33.13.2. Lengthening Instructions

These instructions do a simple operation: the second operand is copied to the first operand register and extended with zeros. The Condition Code is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED24	LDE	RXE	Load Lengthened (Short to Long)	B324	LDER	RRE	Load Lengthened (Short to Long)
ED26	LXE	RXE	Load Lengthened (Short to Extended)	B326	LXER	RRE	Load Lengthened (Short to Extended)
ED25	LXD	RXE	Load Lengthened (Long to Extended)	B325	LXDR	RRE	Load Lengthened (Long to Extended)

Table 249. Hexadecimal floating-point Load Lengthened instructions

For example, to lengthen a short operand to long, you could write:

```
LE    0,=X'12345678'     Short hexadecimal floating-point value
LDER  4,0                c(FPR4)=X'1234567800000000'
```

For extended targets (of LX-type instructions), the signs of the first operand's high- and low-order halves are the same, and the low-order characteristic is 14 less than the high-order characteristic (modulo 128).

```
LD    0,=X'123456789ABCDEF0'  Source operand
LXDR  4,0                     Extend in (FPR4,6)
```

will set FPR4 to the same value that is in FPR0, and FPR6 will be set to X'0400000000000000'.

If the source fraction for an extended target is negative zero, the result is all zero digits except for the two matching sign bits. For example:

```
LD    0,=X'8000000000000000'  Minus zero
LXDR  4,0                     Extend in (FPR4,6)
```

will set both FPR4 and FPR6 to X'8000000000000000'.

Bonus effect

LXE and LXD provide an easy way to load a short or long operand into a floating-point register while also setting its paired register to zero.

Exercises

33.13.1.(2)+ Suppose an LRER or LRDR instruction causes an exponent overflow. What are the possible values of the rounded result?

33.13.2.(2)+ Show the result in FPR0 of executing

```
LRER  0,0
```

with each of these register contents:

1. c(FPR0)=X'7FFFFFFF87654321'
2. c(FPR0)=X'4000000087654321'
3. c(FPR0)=X'4000000012345678'

33.13.3.(2) Write and execute instructions to verify that the sequence shown in Figure 396 on page 618 works as described.

33.13.4.(1) Suppose you execute these instructions:

```
LE    4,Variable
LDER  2,4
LEDR  0,2
```

What differences will there be between c(FPR0) and c(FPR4)?

33.13.5.(2) Show the contents of the registers at each step in Figure 396 on page 618.

33.14. Converting Between Binary Integers and HFP

Before these instructions were added to the System z instruction set, converting between hexadecimal floating-point and binary integers was somewhat roundabout. The instructions in Table 250 greatly simplify the process.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3B4	CEFR	RRE	Convert from Fixed (32 to Short)	B3B8	CFER	RRF	Convert to Fixed (Short to 32)
B3B5	CDFR	RRE	Convert from Fixed (32 to Long)	B3B9	CFDR	RRF	Convert to Fixed (Long to 32)
B3B6	CXFR	RRE	Convert from Fixed (32 to Extended)	B3BA	CFXR	RRF	Convert to Fixed (Extended to 32)
B3C4	CEGR	RRE	Convert from Fixed (64 to Short)	B3C8	CGER	RRF	Convert to Fixed (Short to 64)
B3C5	CDGR	RRE	Convert from Fixed (64 to Long)	B3C9	CGDR	RRF	Convert to Fixed (Long to 64)
B3C6	CXGR	RRE	Convert from Fixed (64 to Extended)	B3CA	CGXR	RRF	Convert to Fixed (Extended to 32)

Table 250. Hexadecimal floating-point FPR/GPR conversion instructions

33.14.1. Converting Binary Integers to Hexadecimal Floating-Point

The six instructions in the left-hand columns of Table 250 convert a binary integer in a general register to a hexadecimal floating-point value in a floating-point register. For example, we can convert the integer in GR5 to short hexadecimal floating-point in FPR4:

```
L     5,=F'54321'
CEFR  4,5              c(FPR4)=X'44D43100'
```

Figure 397. Converting a 32-bit integer to short hexadecimal floating-point

The rules for these six instructions are simple:

1. Zero integer values are converted to +0 floating-point values.
2. The HFP result is truncated, not rounded.
3. The Condition Code is unchanged.
4. The CXFR and CXGR instructions require that the R_1 operand refer to the lower-numbered register of a Floating-Point Register pair.

Converting 64-bit integers to hexadecimal floating-point is just as easy, but you can see in Figure 398 that large integer values may lose some low-order bits when converted to short or long hexadecimal floating-point formats:

```
LG    5,=FD'123456789012345678'    (X'01B69B4BA630F34E')
CEGR  4,5      c(FPR4)=X'4F1B69B4'         (truncated)
CDGR  4,5      c(FPR4)=X'4F1B69B4BA630F34' (truncated)
CXGR  4,5      c(FPR4)=X'4F1B69B4BA630F34 41E0000000000000' (OK)
```

Figure 398. Converting a 64-bit integer to three hexadecimal floating-point values

To help you appreciate the simplicity of these instructions, consider this sequence of instructions used by an early System/360 FORTRAN compiler[217] to convert the fullword integer stored at **FWInt** to hexadecimal floating-point, and store the result at **DFInt**.

[217] A *very* early compiler. Subsequent compilers quickly used improved techniques.

```
         LD    0,=D'0'           Clear all of FPR0
         L     1,FWInt           Load integer from FWInt into GPR1
         LTR   1,1               Test sign
         BM    Neg               Branch if negative
         ST    1,DCon+4          Store magnitude
         AD    0,DCon            Normalized result in FPR0
         B     Store             Branch to store
Neg      LPR   1,1               Take magnitude of integer
         ST    1,DCon+4          Store it
         SD    0,DCon            Form negative normalized value
Store    STD   0,DFInt           Store long floating value
         - - -
DFInt    DS    D                 Doubleword result
DCon     DC    X'4E',7X'0'       Pseudo-zero, exponent = 14
```

Figure 399. Early conversion of integer to hexadecimal floating-point

More efficient and elegant techniques were developed as high-level languages evolved; some of them are illustrated in (the recommended!) Exercises 33.14.4 through 33.14.6.

When converting a fullword integer to short hexadecimal floating-point, there may be as many as 32 significant bits in the integer, but a short HFP number has only a 24-bit fraction. Thus, there may be a loss of accuracy in converting fullword integers to short hexadecimal floating-point. Since floating-point arithmetic is concerned primarily with maintaining as many high-order digits as possible, this truncation is tolerated by the new instructions also.

33.14.2. Converting Hexadecimal Floating-Point to Binary Integers

The six instructions in the right-hand columns of Table 250 on page 620 convert a hexadecimal floating-point value in a floating-point register to a binary integer in a general register. Any fractional value is lost, except that rounding the lost fraction might affect the low-order digit of the integer result. In addition to the expected R_1 and R_2 operands, they also specify a rounding modifier M_3 as another operand. The instruction format is:

opcode	M_3		R_1	R_2

Table 251. Format of HFP to fixed binary instructions

Calling it the “third” operand may be confusing, because it is actually specified as the *second* operand of a machine instruction statement:

```
         CFER  R1,M3,R2
```

Figure 400. Format of a machine instruction statement for converting HFP to binary

The R_1 (target) operand specifies a general register and the R_2 (source) operand specifies a floating-point register. The M_3 operand determines what rounding should be done, as shown in Table 252. (These instructions are easy to use; just be sure to specify a rounding mode (M_3) operand, as there is no default.)

M_3	Meaning
B'0000'	Truncate the fraction part (round toward zero)
B'0001'	Round to nearest; if the source value is halfway between two integer values, choose the one with the larger magnitude
B'0100'	Round to nearest; if the source value is halfway between two integer values, choose the even one with a zero low-order bit
B'0101'	Truncate the fraction part (round toward zero)
B'0110'	Round toward $+\infty$
B'0111'	Round toward $-\infty$

Table 252. Rounding modifiers for HFP-to-binary conversion

Other modifier values are invalid, and cause a specification exception.

These instructions set the Condition Code, with CC=3 as an interesting and unusual case.

CC	Meaning
0	Result is = 0
1	Result is < 0
2	Result is > 0
3	The rounded result exceeds the range of the target representation, and has been replaced by the largest representable correctly signed magnitude.

Table 253. CC settings for HFP-to-binary conversion

As you would expect, the CFXR and CGXR instructions require that the R_2 operand reference the low-order half of a floating-point register pair.

For example, these instructions generate the indicated results:

```
LE    4,=E'123.456'
CFER  2,B'0110',4        c(GR2) = X'0000007C' (=124)
LD    4,=E'-73.946'
CGER  2,B'0111',4        c(GG2) = X'FFFFFFFFFFFFFFB6' (=-74)
LE    4,=E'1234567890'
CFER  2,B'0100',4        c(GR2) = X'49960300' (=1234567936)
```

The last example shows that some low-order bits could not be represented in the short second operand.

Unexpected results can also be generated if floating-point values "known" to have integer values are converted to binary. For example if a HFP result is expected to be 8 but actually is calculated as 7.999999, the lack of a rounding conversion to binary generates the integer value 7.

Table 254 summarizes the actions of some instructions that move and/or convert data between and among the general registers and floating-point registers:

To→ ↓ From	**Fixed Bin (32 bits)**	**Fixed Bin (64 bits)**	**Hex Float (32 bits)**	**Hex Float (64 bits)**	**Hex Float (128 bits)**
Fixed Bin (32 bits)	LR	LGFR	CEFR	CDFR	CXFR
Fixed Bin (64 bits)	LR	LGR	CEGR	CDGR	CXGR
Hex Float (32 bits)	CFER	CGER	LER	LDER	LXER
Hex Float (64 bits)	CFDR	CGDR	LRER LEDR	LDR	LXER
Hex Float (128 bits)	CFXR	CGXR	LEXR	LRDR LDXR	LXR

Table 254. Instructions moving/converting binary and hexadecimal floating-point operands

Exercises

33.14.1.(2) An array of fullword integers is stored starting at **IntData** and the number of elements in the array is stored as a fullword integer at **NItems**. Write instructions to compute and store at **IntAvg** the floating-point average of the list of integers, taking into account the possibility that the integer sum may overflow a general register.

33.14.2.(2) What is the largest binary integer that can be converted to hexadecimal floating-point long format without loss of precision? Explain.

33.14.3.(1) Can the instructions for converting a binary integer to hexadecimal floating-point generate an exponent spill?

33.14.4.(3)+ A compiler used these instructions to convert the fullword binary integer in GR0 to long hexadecimal floating-point form in FPR0.

```
         X     0,Float+4          Invert sign bit
         ST    0,DCon+4           Store result in long number
         LD    0,DCon             Load the unnormalized value
         SD    0,Float            Subtract and normalize
         - - -
         DS    0D                 Align on doubleword boundary
DCon     DC    X'4E00000000000000'     Pseudo-zero, exponent = 14
Float    DC    X'4E00000080000000'     2**31, unnormalized
```

Analyze this method, and show (a) how it works, and (b) that it will work correctly for any integer value.

33.14.5.(3) Another compiler used this instruction sequence to convert a fullword binary integer in GR0 to long hexadecimal floating-point form in FPR0. Analyze its action for positive, zero, negative, and maximum negative integer values. Describe the differences between this instruction sequence and that in Exercise 33.14.4.

```
         X     0,DCon+4           Invert sign bit
         ST    0,Float+4          Store result in long number
         LD    0,Float            Load the unnormalized value
         AD    0,DCon             Add and normalize
         - - -
         DS    0D                 Align on doubleword boundary
DCon     DC    X'CE00000080000000'     -2**31 unnormalized
Float    DC    X'4E00000000000000'     pseudo-zero, exponent = 14
```

33.14.6.(3)+ The following instruction sequence was used by a compiler to convert a short hexadecimal floating-point value at **X** to a fullword integer in GR0. Analyze its operation for positive, zero, and negative hexadecimal floating-point values, and describe its behavior for hexadecimal floating-point values exceeding 2^{31}.

```
         LD    0,DCon             Clear low-order half of FPR0
         LE    0,X                Get floating-point operand
         AD    0,DCon             Add and normalize
         STD   0,FTemp            Store the result
         L     0,FTemp+4          Put the integer result in GR0
         - - -
FTemp    DS    D                  Doubleword temporary
DCon     DC    X'4F08000000000000'     Conversion constant
```

Explain the form of the constant at **DCon**.

33.14.7.(3) What modifications would be required to the instructions in Exercise 33.14.6 to handle conversion of a long precision hexadecimal floating-point value to fullword integer form? To 64-bit integer form?

33.14.8.(2)+ Write the constants named **DCon** in Exercises 33.14.4 through 33.14.6 as hexadecimal floating-point constants in DC statements.

33.14.9.(2)+ What integer values satisfying

$2^{30} \le \text{value} < 2^{31}$

can be converted without loss of precision to short hexadecimal floating-point form?

33.14.10.(3) Without using the instructions described in Section 33.14, write instructions to convert the long hexadecimal floating-point value at **YY** to a fullword binary integer stored at

NN. If the integer result is too large to be correctly represented, branch after storing the result to the instruction at location **TooBig**.

33.14.11.(3) Without using the instructions described in Section 33.14, write instructions to convert the short hexadecimal floating-point number at **XX** to a correctly rounded fullword integer value stored at **MM**.

33.14.12.(2) Write the constant named **Float** in Exercise 33.14.4 in a DC statement as a scaled D-type constant.

33.14.13.(3)+ Write an instruction sequence that will convert the nonnegative fullword integer at **IntVal** to a rounded short hexadecimal floating-point format stored at **Result**, using round half-up (arithmetic) rounding.

33.14.14.(3) Repeat Exercise 33.14.13, but use half-even rounding.

33.14.15.(2) The following short HFP numbers are converted to binary integers using these two instructions:

```
(a)      LE    0,HFP_Number
         CFER  3,0,0

(b)      LE    0,HFP_Number
         CGER  3,0,0
```

for these HFP_Number values:

(1) X'46000000'
(2) X'C7654321'
(3) X'7FEDCB49'
(4) X'ABCDEF74'
(5) X'4974662B'

Show the Condition Code and the contents of GR3 and GG3 for each HFP number.

33.14.16.(2)+ If you convert the HFP constant E'65537' to a halfword binary integer, what is the result?

33.14.17.(2) Suppose you use these instructions to convert a long HFP operand at **DFloat** to a fullword binary integer stored at **NF**. What value will be stored?

```
         LD    0,DFloat          Get long operand
         AW    0,DCon            Add long pseudo-zero, exponent 14
         STD   0,DTemp           Store temporarily
         L     1,DTemp+4         Pick up fullword integer
         BNM   *+6               Branch if nonnegative
         LCR   1,1               Complement integer value
         ST    1,NF              Store fullword result
         - - -
DTemp    DS    D
DCon     DC    X'4E',7X'0'       Pseudo-zero, exponent=14
DFloat   DC    D'16777219'       Long HFP source data
NF       DS    F                 Integer result
```

33.15. Hexadecimal Floating-Point Integers and Remainders (*)

We must sometimes compute the "remainder" of a floating-point division, even though it's not provided by the CPU as a result of any divide instruction. Given two operands A and B, we may need to calculate a quotient Q and a remainder R so that

```
A = Q*B + R
```

The quotient Q is produced by the divide instructions; for floating-point arithmetic, R must be calculated separately. This can be done using

```
R = A - Int(A/B)*B
```

Thus, we evaluate A/B, convert it to its floating-point integer form, multiply that by B, and subtract the product from A to give the desired result.

For example, suppose we use three-digit decimal floating-point and divide 5.55 by 2.47; the true quotient is 2.246963562753.... The integer part of the quotient is 2, so we must evaluate $5.55-(2\times2.47)$ to find the remainder, 0.61.

The key step is finding the integer part of the quotient, using the instructions in Table 255.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B377	FIER	RRE	Load FP Integer (Short)	B37F	FIDR	RRE	Load FP Integer (Long)
B367	FIXR	RRE	Load FP Integer (Extended)				

Table 255. Hexadecimal floating-point instructions generating floating-point integers

It's important to remember that the result is a *hexadecimal floating-point integer* in the same format, not a fixed binary integer!

- The results of these instructions are normalized.
- The Condition Code is unchanged.
- FIXR requires that both operands refer to the lower-numbered register of a floating-point register register pair.
- If the exponent of the second operand is large enough, the result in the first operand will be the same value (normalized).

Using the values A=5.55 and B=2.47, Figure 401 shows how we would calculate the HFP remainder:

```
LE    0,=E'5.55'         A = 5.55
DE    0,=E'2.47'         B = 2.47
FIER  0,0                Int(A/B) = 2
ME    0,=E'2.47'         Multiply by B
LCER  0,0                Invert sign of the product
AE    0,=E'5.55'         Remainder = A - Int(A/B)*B
```

Figure 401. Calculating a HFP remainder

The result in FPR0 is X'409C28F0', or approximately 0.60999966 in decimal.

To give another example:

```
LE    0,=X'42345678'     Hex 33.5678
FIER  2,0                c(FPR2) = X'4234000'
```

removes the fraction X'42005678', leaving the integer part.

Suppose there are short hexadecimal floating-point operands at **A** and **B** and the remainder is to be stored in short form at **AModB**.

```
         LE    0,A                 Load A into FPR0
         DE    0,B                 Divide by B
         FIER  0,0                 Drop off the fraction part
         ME    0,B                 Form product with integer part
         LCER  0,0                 Form -B*IntPart(A/B)
         AE    0,A                 Add A to form remainder
         STE   0,AModB             Store result
         - - -
AModB    DS    E                   Remainder
A        DC    E'20'               A value for A
B        DC    E'16'               And for B
```

Figure 402. Evaluating a hexadecimal floating-point remainder

This technique should be used with care. If the relative magnitudes of A and B are close, the result is acceptable; but if the ratio |A/B| approaches the precision of the operands, the result can be inaccurate, and special programming techniques may be needed to calculate an accurate result.

Exercises

33.15.1.(2)+ In Figure 402, what will be in FPR0 after the DE instruction? After the FIER instruction? After the ME instruction? After the AE instruction?

33.15.2.(2)+ Show that the FIER instruction in Figure 402 can be replaced by

```
         AU    0,=X'46000000'
```

33.15.3.(3) What will happen in Figure 402 if the FIER instruction is replaced by

```
         AE    0,=X'46000000'      ?
```

For what values of A/B will there be no difference between using AE and AU instructions?

33.15.4.(1) Rewrite the instructions in Figure 402 to use long operands.

33.15.5.(2) What is the result in FPR0 of each of these instruction sequences?

```
(1)      LE    0,=X'44001234'
         FIER  0,0

(2)      LD    0,=X'C7FEDCA987654321'
         FIDR  0,0

(3)      LXD   0,=X'1234567890ABCDEF'
         FIXR  0,0

(4)      LE    0,=X'77654321'
         FIER  0,0

(5)      LD    0,=X'C7FEDCA987654321'
         FIER  0,0
```

33.16. Square Root Instructions (*)

The instructions in Table 256 on page 627 extract the square root of a hexadecimal floating-point operand. None of them change the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED34	SQE	RXE	Square Root (Short)	B245	SQER	RRE	Square Root (Short)
ED35	SQD	RXE	Square Root (Long)	B244	SQDR	RRE	Square Root (Long)
				B336	SQXR	RRE	Square Root (Extended)

Table 256. Hexadecimal floating-point Square Root instructions

The SQXR instruction requires both the R_1 and R_2 operands to refer to the lower-numbered register of a floating-point register pair.

The second operand is first tested to see if it is negative and nonzero. If so, a HFP Square Root interruption occurs, and the IC is set to 29 (X'1D'). Otherwise, the operand is normalized internally and its square root is evaluated; the result is normalized, rounded, and its sign is always +.

Because the exponent of a square root is about one-half the exponent of its argument, the result cannot cause an exponent spill.

Figure 403 gives some examples of these instructions:

```
LE    2,=E'1024.576'          c(FPR2)=X'43400937'
SQER  0,2                     c(FPR0)=X'4220024E' = 32.00900...
LD    2,=X'41FFFFFFFFFFFFFF'  = 15.999999999997...
SQDR  0,2                     c(FPR0)=X'4140000000000000' = 4.0
LE    2,=X'80000000'          Minus zero
SQER  0,2                     c(FPR0)=X'00000000'
```

Figure 403. Examples of HFP square root instructions

These instructions have two interesting properties (see Exercises 33.16.2 and 33.16.3):

1. Rounding cannot produce a carry out of the high-order digit of the result.
2. A square root cannot lie exactly half way between two representable values, so that traditional “half-up” rounding can be used.

Exercises

33.16.1.(1)+ Estimate the value of the square root of HFP (Max), (Min), and (DMin) values.

33.16.2.(3) It is stated above that rounding the result of a square root instruction cannot create a carry out of the high-order digit of the fraction. Why not?

33.16.3.(4) It is stated above that the result of a square root instruction cannot lie exactly half way between two representable values. Why not?

33.17. Multiply and Add/Subtract Instructions (*)

The instructions in Table 257 combine two operations into one: a multiplication followed by an addition or subtraction; they are sometimes called “fused multiply-add” or “FMA” instructions. None of them change the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED2E	MAE	RXF	Multiply and Add (Short)	B32E	MAER	RRF	Multiply and Add (Short)
ED3E	MAD	RXF	Multiply and Add (Long)	B33E	MADR	RRF	Multiply and Add (Long)
ED2F	MSE	RXF	Multiply and Subtract (Short)	B32F	MSER	RRF	Multiply and Subtract (Short)
ED3F	MSD	RXF	Multiply and Subtract (Long)	B33F	MSDR	RRF	Multiply and Subtract (Long)

Table 257. Hexadecimal floating-point Multiply and add/subtract instructions

These instructions have three operands; their RRF format (shown in Table 258) is similar to that in Table 251 on page 621, except that the instruction operands are in different fields.

opcode	R_1		R_3	R_2

Table 258. Format of RRF-type HFP multiply and add/subtract instructions

The RXF-type instruction format is shown in Table 259:

opcode	R_3	X_2	B_2	D_2	R_1		opcode

Table 259. Format of RXF-type multiply and add/subtract instructions

The assembler instruction statement format for these instructions is simpler:

```
MAE   R1,R3,D2(X2,B2)      Explicit address
MAE   R1,R3,S2             Implied address
MAER  R1,R3,R2
```

The effect of these instructions is to evaluate

$$\text{operand_1} = (\text{operand_3} \times \text{operand_2}) \pm \text{operand_1}$$

or

$$c(R_1) = (c(R_3) \times \text{operand_2}) \pm c(R_1)$$

Initially, the product of operand_2 and operand_3 is evaluated internally to double precision: that is, to 14 digits for short operands, and to 28 digits for long operands. Possible exponent spills are ignored at this stage. Then, operand_1 is added or subtracted, the result is normalized and truncated, and replaces operand_1. Exponent overflow or underflow may occur at this stage.

A single guard digit is used for the addition or subtraction, and the final result is truncated, not rounded.

None of the operands is normalized at the start of the instructions.

For example:

```
LE    4,=E'25'          Operand_3 in FPR4
LE    2,=E'123'         Operand_1 in FPR2
MAE   2,4,=E'4'         Operand_2 = 4.0
```

multiplies 25 and 4, and adds 123, giving X'42DF0000', or 223 as expected.

The big advantage of these instructions is that they avoid the possibility of double rounding, or the need for using higher-precision arithmetic. If you wanted to achieve the same result without the MAE instruction, you would have to write something like this:

```
LE    4,=E'25'          Operand_3
LDER  4,4               Extend to long precision
LE    2,=E'4'           Operand_2
LDER  2,2               Extend to long precision
MDR   4,2               Form long product
LE    2,=E'123'         Operand_1
LDER  2,2               Extend to long precision
ADR   2,4               Accumulate final result
```

This not only takes more instructions (and probably executes more slowly), but it is open to the possibility that the MDR instruction could cause an exponent spill that would otherwise have been compensated by the ADR instruction. For example:

```
LE    4,=X'60987654'    Operand_3
LE    2,=X'FFFFFFFF'    Operand_1
MAE   2,4,=X'60234567'  Operand_2 in memory
```

produces a result X'7F50182C' in FPR2 with no overflow, but

```
         LE    4,=X'60987654'      Operand_3
         ME    4,=X'60234567'      Operand_2 in memory
         AE    4,=X'FFFFFFFF'      Operand_1
```

produces an exponent overflow when the ME instruction is executed with c(FPR4)=X'00150182B83FCC00'; the following AE instruction ignores the (now) tiny result of the ME instruction, giving X'FFFFFFFF' as the final result!

Exercises

33.17.1.(2)+ Rewrite the example in Figure 387 on page 608 to use multiply-add instructions.

33.17.2.(2) Rewrite the example in Figure 388 on page 608 to use multiply-add instructions.

33.18. Some Hexadecimal Floating-Point History (*)

In the early System/360 CPUs, processor and memory speeds were close (and in some cases a memory access was faster than the time needed to execute an instruction). Some programming techniques that performed acceptably then are no longer used.

33.18.1. Zeroing Floating-Point Registers

Sometimes you need to clear a floating-point register to zero, or set the low-order half of a floating-point register to zero prior to loading a short operand. Without the lengthening instructions in Section 33.13.2 on page 618, or the Load Zero instructions in Section 31.9.3. on page 569, you might have used instructions like these:

```
         LD    0,=D'0.0'           Load a zero from memory
or
         SWR   0,0                 Subtract the register from itself
```

with the cost of a constant and a memory access for the LD instruction, and a possible HFP lost-significance exception for the SWR instruction.

33.18.2. Hexadecimal Floating-Point to Binary Conversion Comments (*)

There are some possible problems.

1. Suppose we try to convert the HFP constant E'50000' to a halfword integer. If we use a typical instruction sequence, the intermediate HFP result would be X'4600C350'. If we load the rightmost four digits of this result into a general register using a LH instruction, the contents of that register would be X'FFFFC350', an integer value of −15536! This could be quite a surprise, because the original operand was positive. Such conversions gave no indication that the result was corrupted.

   While this example might seem relatively harmless (the negative result indicates that an overflow has occurred), it is easy to create examples where the loss of accuracy is not as obvious.

2. The lack of a CC setting meant that the binary result would have to be tested using a LTR instruction to determine its sign; the new instructions like those in Section 33.14.2 on page 621 solve that problem

3. When converting an integer to short hexadecimal floating-point, it is possible that the integer has enough low-order zero bits so that its magnitude may exceed 2^{24} and still no significant bits will be lost in the conversion, but we have no way to know this is so. While this may not seem very important, there are situations where you may obtain unexpected results.

33.18.3. Initial System/360 Oversights

When the very first System/360 CPUs were delivered, customers found that some floating-point implementation oversights made program portability more difficult among different models:

1. There was no guard digit for long hexadecimal floating-point arithmetic. (For example, the multiplicative identity 1×a=a failed in long arithmetic due to the lack of a guard digit.)
2. No post-normalization or zero-fraction test was performed after the Halve instructions, so that unnormalized results could be generated from normalized operands.
3. No characteristic wraparound or correct fraction was guaranteed in case of overflow and underflow, and the result left in the register depended on the model.
4. A condition code of 3 was generated when exponent overflow occurred during add operations.
5. No extended precision operations were available.

These oversights were corrected very promptly: IBM announced in February 1967 that engineering changes would be made to all machines (if desired by the owner or renter) in order to rectify the first four of the above conditions. Extended precision was delivered several years later.

When IBM made the announcement at the SHARE conference in February 1967, the speaker asked "How many of you have System/360 Model 30 machines?" Almost all the attendees' hands went up. He then asked "How many of you would be willing to forgo the hardware updates?" No hands went up. He then said "Well, I promised I'd ask."

The reason for the question was that IBM had shipped a very large number of Model 30 processors, and the required microcode updates were quite extensive (and costly to IBM).

33.19. Summary

We've covered a lot of material in this section; a few summary comments may help.

- Operands are pre-normalized for multiplication and division, but not for addition or subtraction.
- There are no reserved or special values, as for binary and decimal floating-point.
- Arithmetic results are not rounded.
- Arithmetic truncates (usually, *but not always*, toward zero!) For example: `X'42100000'−X'40FFFFFF' = X'41F00001'` (instead of `X'41F00000'`); this result is truncated *away* from zero, with error = 15/16 ulp.
- Zero arithmetic results are delivered as +0 ("true zero").

There are no instructions operating on two hexadecimal floating-point operands of different lengths, such as adding a short and a long operand. Such arithmetic is sometimes called "mixed-length" or "mixed-precision" arithmetic. The only practical way to mix short and long operands is to clear the low-order half of a long register (or area of memory), place the short operand in the high-order half, and then do the required operation in long arithmetic.

The hexadecimal floating-point instructions for moving and testing data in the floating-point registers are summarized in Table 260 on page 631.

Function	Operand Length		
	4 bytes	8 bytes	16 bytes
Load and Test	LTER	LTDR	LTXR
Load Complement	LCER	LCDR	LCXR
Load Negative	LNER	LNDR	LNXR
Load Positive	LPER	LPDR	LPXR

Table 260. Hexadecimal floating-point Move/Test instructions

The hexadecimal floating-point multiplication instructions are summarized in Table 261.

Func-tion	Source Length	4 bytes		8 bytes		16 bytes
	Product Length	4 bytes	8 bytes	8 bytes	16 bytes	16 bytes
Multiply (registers)		MEER	MER MDER	MDR	MXDR	MXR
Multiply (storage)		MEE	ME MDE	MD	MXD	

Table 261. Hexadecimal floating-point Multiply instructions

The hexadecimal floating-point divide instructions are summarized in Table 262.

Function	Operand Length		
	4 bytes	8 bytes	16 bytes
Divide (register)	DER	DDR	DXR
Divide (storage)	DE	DD	
Halve (register)	HER	HDR	

Table 262. Hexadecimal floating-point Divide instructions

The hexadecimal floating-point instructions for normalized and unnormalized addition and subtraction and for comparison are summarized in Table 263.

Function	Operand Length		
	4 bytes	8 bytes	16 bytes
Add/Subtract Normalized (register)	AER SER	ADR SDR	AXR SXR
Add/Subtract Unnormalized (register)	AUR SUR	AWR SWR	
Add/Subtract Normalized (storage)	AE SE	AD SD	
Add/Subtract Unnormalized (storage)	AU SU	AW SW	
Compare (register)	CER	CDR	CXR
Compare (storage)	CE	CD	

Table 263. Hexadecimal floating-point Add, Subtract, and Compare instructions

The hexadecimal floating-point rounding instructions are summarized in Table 264 on page 632.

Function	Source Length	8 bytes	16 bytes	16 bytes
	Result Length	4 bytes	4 bytes	8 bytes
Round		LRER LEDR	LEXR	LRDR LDXR

Table 264. Hexadecimal floating-point Round instructions

The hexadecimal floating-point operand-lengthening instructions are summarized in Table 265.

Function	Source Length	4 bytes		8 bytes
	Result Length	8 bytes	16 bytes	16 bytes
Lengthen (register)		LDER	LXER	LXDR
Lengthen (storage)		LDE	LXE	LXD

Table 265. Hexadecimal floating-point Lengthening instructions

The instructions for converting hexadecimal floating-point operands to binary integers are summarized in Table 266.

Function	Source Length	4 bytes		8 bytes		16 bytes	
	Target Length	32 bits	64 bits	32 bits	64 bits	32 bits	64 bits
Convert float to binary		CFER	CGER	CFDR	CGDR	CFXR	CGXR

Table 266. Convert hexadecimal floating-point to binary instructions

The instructions for converting binary integer operands to hexadecimal floating-point are summarized in Table 267.

Function	Source Length	32 bits			64 bits		
	Target Length	4 bytes	8 bytes	16 bytes	4 bytes	8 bytes	16 bytes
Convert binary to float		CEFR	CDFR	CXFR	CEGR	CDGR	CXGR

Table 267. Convert binary to hexadecimal floating-point instructions

The instructions for removing the fraction part of hexadecimal floating-point operands (that is, extracting the integer portion) are summarized in Table 268.

Function	Operand Length	4 bytes	8 bytes	16 bytes
Form floating-point integer		FIER	FIDR	FIXR

Table 268. Form hexadecimal floating-point integer instructions

The instructions for evaluating the square root of hexadecimal floating-point operands are summarized in Table 269.

Function	Operand Length		
	4 bytes	8 bytes	16 bytes
Square root (register)	SQER	SQDR	SQXR
Square root (register)	SQE	SQD	

Table 269. Hexadecimal floating-point Square Root instructions

The instructions for performing multiply-and-add and multiply-and-subtract operations on hexadecimal floating-point operands are summarized in Table 270 on page 633.

Function	Operand Length	
	4 bytes	8 bytes
Multiply-add (register)	MAER	MADR
Multiply-add (storage)	MAE	MAD
Multiply-subtract (register)	MSER	MSDR
Multiply-subtract (storage)	MSE	MSD

Table 270. Hexadecimal floating-point Multiply-Add/Subtract instructions

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
AD	6A
ADR	2A
AE	7A
AER	3A
AU	7E
AUR	3E
AW	6E
AWR	2E
AXR	36
CD	69
CDFR	B3B5
CDGR	B3C5
CDR	29
CE	79
CEFR	B3B4
CEGR	B3C4
CFDR	B3B9
CFXR	B3BA
CGDR	B3C9
CGER	B3C8
CGXR	B3CA
CXFR	B3B6
CXGR	B3C6
CER	39
CFER	B3B8
CXR	B369
DD	6D
DDR	2D
DE	7D
DER	3D
DXR	B22D

Mnemonic	Opcode
FIDR	B37F
FIER	B377
FIXR	B367
HDR	24
HER	34
LCDR	23
LCER	33
LCXR	B363
LDE	ED24
LDER	B324
LDXR	25
LEDR	35
LEXR	B366
LNDR	21
LNER	31
LNXR	B361
LPDR	20
LPER	30
LPXR	B360
LRDR	25
LRER	35
LTDR	22
LTER	32
LTXR	B362
LXD	ED25
LXDR	B325
LXE	ED26
LXER	B326
MAD	ED3E
MADR	B33E
MAE	ED2E

Mnemonic	Opcode
MAER	B32E
MD	6C
MDE	7C
MDER	3C
MDR	2C
ME	7C
MEE	ED37
MEER	B337
MER	3C
MSD	ED3F
MSDR	B33F
MSE	ED2F
MSER	B32F
MXD	67
MXDR	27
MXR	26
SD	6B
SDR	2B
SE	7B
SER	3B
SQD	ED35
SQDR	B244
SQE	ED34
SQER	B245
SQXR	B336
SU	7F
SUR	3F
SW	6F
SWR	2F
SXR	37

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
20	LPDR
21	LNDR
22	LTDR
23	LCDR
24	HDR
25	LDXR
25	LRDR
26	MXR
27	MXDR
29	CDR
2A	ADR
2B	SDR
2C	MDR
2D	DDR
2E	AWR
2F	SWR
30	LPER
31	LNER
32	LTER
33	LCER
34	HER
35	LEDR
35	LRER
36	AXR
37	SXR
39	CER
3A	AER
3B	SER
3C	MDER
3C	MER
3D	DER

Opcode	Mnemonic
3E	AUR
3F	SUR
67	MXD
69	CD
6A	AD
6B	SD
6C	MD
6D	DD
6E	AW
6F	SW
79	CE
7A	AE
7B	SE
7C	MDE
7C	ME
7D	DE
7E	AU
7F	SU
B22D	DXR
B244	SQDR
B245	SQER
B324	LDER
B325	LXDR
B326	LXER
B32E	MAER
B32F	MSER
B336	SQXR
B337	MEER
B33E	MADR
B33F	MSER
B360	LPXR

Opcode	Mnemonic
B361	LNXR
B362	LTXR
B363	LCXR
B366	LEXR
B367	FIXR
B369	CXR
B377	FIER
B37F	FIDR
B3B4	CEFR
B3B5	CDFR
B3B6	CXFR
B3B8	CFER
B3B9	CFDR
B3BA	CFXR
B3C4	CEGR
B3C5	CDGR
B3C6	CXGR
B3C8	CGER
B3C9	CGDR
B3CA	CGXR
ED24	LDE
ED25	LXD
ED26	LXE
ED2E	MAE
ED2F	MSE
ED34	SQE
ED35	SQD
ED37	MEE
ED3E	MAD
ED3F	MSD

Exercises

33.19.1.(2)+ Suppose you evaluate `X'42100000'−X'40FFFFFF'` using SE and SU instructions. Estimate the size of the error for each result, and explain their difference.

33.19.2.(4) Suppose the designers of System/360 had chosen an octal (base 8) representation for floating-point arithmetic. Consider two choices for the width of a binary characteristic: 7 bits (with 8 fraction digits), or 4 bits (with 9 fraction digits); the remaining bit is for the sign.

Determine the exponent ranges and fraction accuracy for each.

The following six exercises use the definition of ulp(x) described in Section 32.8 on page 580.

33.19.3.(2)+ Suppose a short hexadecimal floating-point number is stored at **X**. Write instructions to calculate ulp(X) and store the result at **ULPX**. Assume that the value at **X** is large enough that no exponent underflow occurs in calculating the result.

33.19.4.(3) What is the minimum value of the hexadecimal floating-point number at **X** that will not underflow when you calculate ulp(X)?

33.19.5.(2)+ Repeat Exercise 33.19.3 assuming that the value at **X** is a long precision hexadecimal floating-point number.

33.19.6.(3) Repeat Exercise 33.19.4 assuming that the value at **X** is a long precision hexadecimal floating-point number.

33.19.7.(3)+ Repeat Exercise 33.19.3 assuming that the value at **X** is an extended precision hexadecimal floating-point number.

33.19.8.(3) Repeat Exercise 33.19.4 assuming that the value at **X** is an extended precision hexadecimal floating-point number.

Terms and Definitions

multiply and add/subtract
An instruction in which a double-length product is created internally to which a third operand is added or subtracted before truncating or rounding the result to the length of the original operands.

rounding-mode suffix
The letter R and a number appended to the numeric value of a constant to request the Assembler to perform a specific form of rounding.

significance exception
In hexadecimal floating-point, the result of an addition or subtraction yields a significand of all zero digits. This exception can either cause a program interruption with IC=14, or can be masked to produce a true zero.

unnormalized add/subtract
Hexadecimal floating-point addition and subtraction in which the result is not normalized.

Programming Problems

Problem 33.1.(2) Write a program to compute and print the hexadecimal floating-point values of the expression

$$f(x) = 1/(x^3 - 3x - 2)$$

for integral values of x from −5 to +5. Print the largest possible hexadecimal floating-point value if the denominator is zero for any value of x in the range.

Problem 33.2.(2) Write a program to compute a table of factorials from 0 to 55, and determine which values will be stored exactly (with no truncation error) for both short and long precision.

Problem 33.3.(4) Write a program to convert short hexadecimal floating-point numbers to a printable decimal format such as "-.123456E+21". Test your program by reading records with eight hexadecimal characters: convert them to a word representing the short floating-point number; convert those values, and print a line showing the original hexadecimal string and the converted result.

Some sample input values are

```
80000000
451E240C
40100000
00000001
7FFFFFFF
```

and the output from the last of these could look like this:

```
 X'7FFFFFFF' = +.723701E+76
```

Problem 33.4.(3) For values of X running from +2.0 to −2.0 in steps of 0.1, write an instruction sequence to compute and print (in short hexadecimal floating-point) the value of the expression

$$7X^4 - 3X^3 + 4x^2 + 5x - 2.$$

Evaluate the expression using only four multiplications by factoring it into "nested" form.

Problem 33.5.(3) Write a program using short, long, and extended precision hexadecimal floating-point arithmetic that shows that the expression (1.0/N)×N=1.0 is true for values of N between 1 and 100 only when N is a power of 2.

Problem 33.6.(3) Using the iterative technique described in Section 31.3.1 on page 554, write a program to evaluate the square root of 2 to 10 significant digits using long hexadecimal floating-point arithmetic, and without using hexadecimal square-root instructions. Format and print the result as a fixed-point value.

Problem 33.7.(2)+ To explore the relationship between significance and accuracy described in Section 25.10, write a program to assemble the 50 decimal values

```
0.062499991
0.062499992
    - - -
0.062499999
0.062500000
0.062500001
    - - -
0.062500040
```

in short and long precision, and observe the behavior of the assembled constants. See if you can find other sequences of decimal values that exhibit similar behavior.

34. Binary Floating-Point Data and Operations

```
 3333333333          44
333333333333        444
33        33       4444
          33      44 44
          33     44  44
       3333     44   44
       3333    44444444444
          33  444444444444
          33         44
33        33         44
333333333333         44
 3333333333          44
```

This section describes binary floating-point data and arithmetic, often known as "IEEE Binary Floating-Point"[218] Its design is based on many years of experience with irregular floating-point architectures that caused great problems in creating stable and reliable software. Binary floating-point was carefully designed[219], has been widely adopted, and is now used on almost all processors. Its benefits include:

- Greater precision and exponent range than hexadecimal floating-point (except for short precision, which has greater exponent range), making exponent spills less likely.
- Rounding for all arithmetic operations.
- Gradual underflow.
- Special values (such as infinity and "Not a Number" or "NaN").
- Ways to control and test for exception conditions.

First, we'll examine the binary floating-point data representations.

34.1. Binary Floating-Point Data

Table 271 summarizes the three binary floating-point data formats supported by System z:

Length (bytes)	Char. (bits)	Min exp.	Max exp.	Char. Bias	Precision	Max Norm. (Max)	Min Norm. (Min)	Min Denorm. (DMin)
4	8	−126	+127	+127	24	$3.4\times10^{+38}$	1.2×10^{-38}	1.4×10^{-45}
8	11	−1022	+1023	+1023	53	$1.8\times10^{+308}$	2.2×10^{-308}	4.9×10^{-324}
16	15	−16382	+16383	+16383	113	$1.2\times10^{+4932}$	3.4×10^{-4932}	6.5×10^{-4966}

Table 271. Binary floating-point data representations

[218] IEEE stands for the Institute of Electric and Electronic Engineers, which publishes standards used in many industries.

[219] The standards committee paid "...meticulous attention to details that hardly matter to most people but matter, when they do matter, very much." (W. Kahan)

Unlike hexadecimal floating-point, the characteristic field width is different for each data length, and certain characteristic bit patterns represent the special values infinity and "Not a Number". Another key difference is that almost all finite values are normalized. This means:

- there are no redundant representations of finite values, and
- the leading significant fraction digit of a normalized number must be 1, so it can be omitted and its presence can be assumed.

Fraction and Significand

Unlike hexadecimal floating-point where all the significant bits of a number (its significand) are present in the fraction field, the fraction of a normal binary floating-point number does *not* contain all its significant bits.

34.1.1. Data Representations

Figure 404 illustrates the three data formats. The three components of each format are

s The sign bit (0 = +, 1 = −)

char
The characteristic ("biased exponent"), with values from C_{min} = 0 to C_{max} = all 1-bits.

fraction
The remaining bits of the significand. Because there is an extra *implied* high-order 1-bit for normal values, significands actually have 24-bit, 53-bit, and 113-bit precisions respectively.

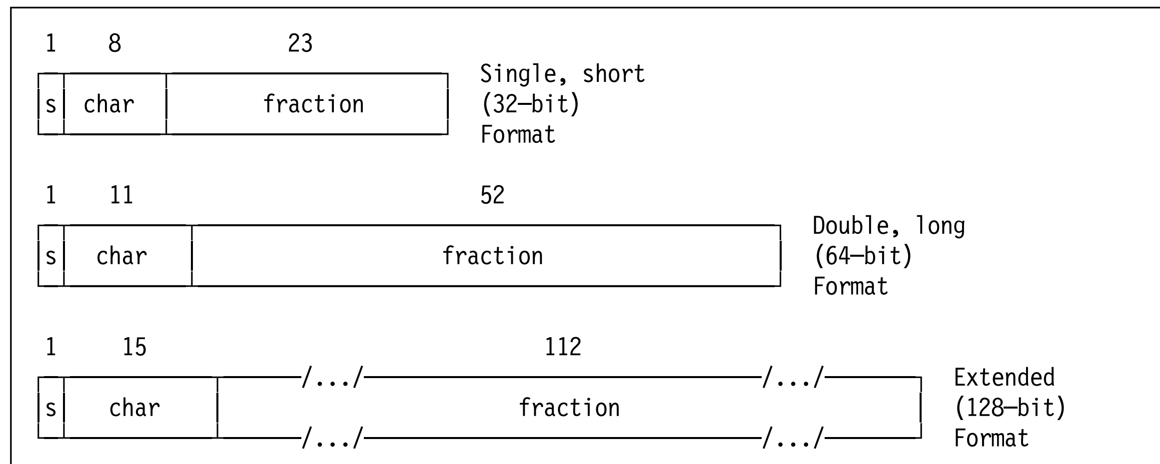


Figure 404. Three binary floating-point data representations

There are six *classes* of binary floating-point data; all are signed:

1. Zero
2. Normal numbers
3. Denormalized numbers
4. Infinity
5. Quiet NaN
6. Signaling NaN

The "Not a Number" (or "Not any Number") was introduced with the binary floating-point standard.[220] It is used to indicate the result of an operation that can't deliver a valid result, such as a division by zero. NaNs are very useful: they let you detect unusable results in your computations when they occur, and with proper error handling you can avoid doing lengthy calculations only to produce useless results.

[220] Some earlier processors supported reserved values such as "indeterminate".

All but the normal numbers are considered special values, even though we normally think of only NaNs and infinity as being "special". Valid results can be (but might not be) generated from data of all classes except NaNs.

Unlike hexadecimal and decimal floating-point, binary floating-point has *no* redundant representations.

34.1.2. Normal Numbers

The exponent of a normal number is determined from its unsigned characteristic:

Characteristic = Exponent + Bias

Two characteristic values, C_{min} (0) and C_{max} (all 1-bits), are reserved for special values. This means that the characteristic's range is $(C_{min} - Bias) + 1 = E_{min}$ to $(C_{max} - Bias) - 1 = E_{max}$, so the exponent range is E_{min} to E_{max}, as shown in Table 271 on page 638.

Because the significand is actually 1.fraction (the leading 1-bit is implied), the fraction is necessarily always normalized, and the significand satisfies $1 \leq$ significand < 2. Thus, the value of a normal number is:

- Single precision (short) value: $\pm(1.\text{fraction}) \times 2^{\text{char}-127}$
- Double precision (long) value: $\pm(1.\text{fraction}) \times 2^{\text{char}-1023}$
- Extended precision (16 bytes) value: $\pm(1.\text{fraction}) \times 2^{\text{char}-16383}$

Table 272 shows examples of short precision normal values.

Value	S	Char.	Fraction	Representation
+1.0	0	01111111	000........000	X'3F800000'
+15.0	0	10000010	1110000....000	X'41700000'
+0.1	0	01111011	10011001...101	X'3DCCCCCD'
+Max	0	11111110	111........111	X'7F7FFFFF'
−Min	1	00000001	000........000	X'80800000'
−0.0	1	00000000	000........000	X'80000000'

Table 272. Examples of short-precision binary floating-point normal values

The first number has characteristic value 127 and zero fraction, so its value is $(1.0) \times 2^{127-127} = 1.0$.

34.1.3. Special Values

Special values are indicated by the reserved characteristic values C_{min} (zero), and C_{max} (all 1-bits).

1. If the characteristic is zero, there are two possible representations:
   - If the fraction is zero, the value is ± 0.
   - If the fraction is not zero, the value is a *denormalized number*.

     In this case, there is no implied 1-bit before the fraction, so the value is $\pm(0.\text{fraction}) \times 2^{E_{min}}$. ($E_{min}$ values are shown in Table 271 on page 638.)

     Table 273 shows examples of short precision denormalized values.

Value	S	Char.	Fraction	Representation
1×10^{-40}	0	00000000	00...011000010	X'000116C2'
Largest Denorm	0	00000000	111........111	X'007FFFFF'
Smallest Denorm	0	00000000	000........001	X'00000001'

Table 273. Examples of short-precision binary floating-point denormalized values

2. If the characteristic is all 1-bits, two special values are represented:
   - If the fraction is zero, the value is ±infinity. We'll see that the sign of an infinity is sometimes important.
   - If the fraction is nonzero, the value is a "NaN". There are two types of NaN:
     – If the leftmost bit of the fraction is 1, the NaN is called a *Quiet NaN*.
     – If the leftmost bit of the fraction is 0, the NaN is called a *Signaling NaN*. Signaling NaNs can cause exceptions.

   The sign of a NaN is ignored.

   Table 274 shows examples of short precision special values:

Value	S	Char.	Fraction	Representation
−Infinity	1	11111111	0000....0000	X'FF800000'
a Quiet NaN	0	11111111	1100....0000	X'7FE00000'
a Signaling NaN	0	11111111	0100....0000	X'7FA00000'

Table 274. Examples of short-precision binary floating-point special values

34.1.4. Range of the Representation

The full range of representable binary floating-point data is sketched in Figure 405:

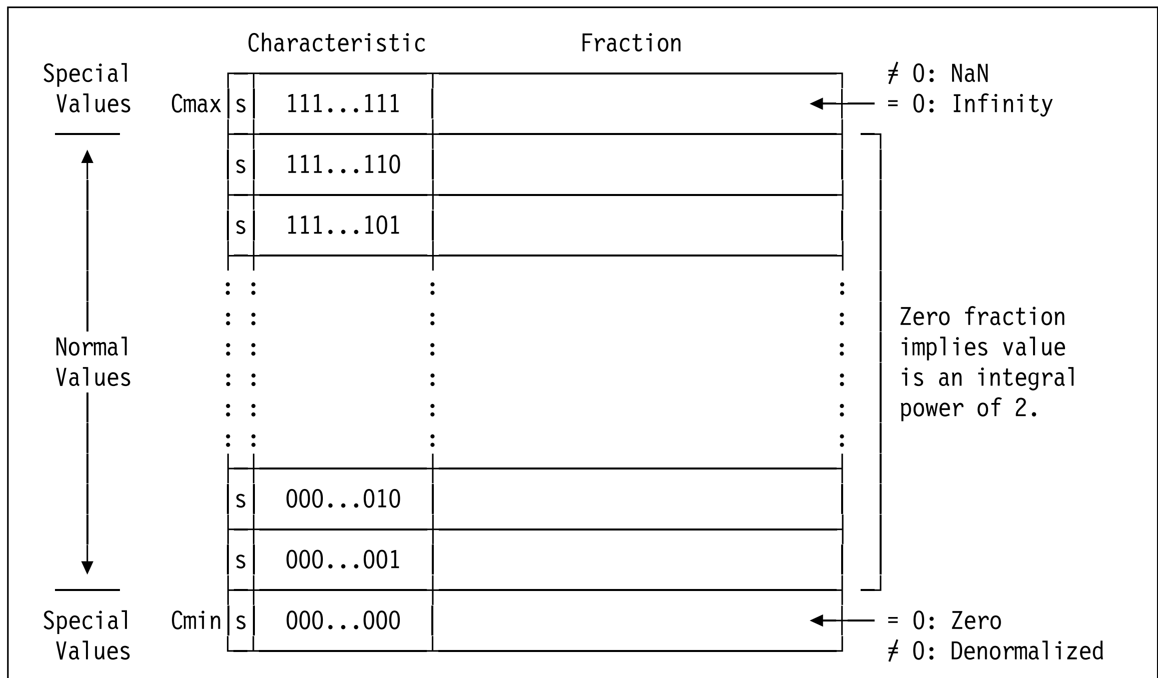


Figure 405. Range of the binary floating-point representation

The special values with characteristic C_{max} are infinity and NaNs; those with characteristic C_{min} are denormalized numbers and zero.

Another view of the binary floating-point representation for computationally valid values is shown in Figure 406 on page 642:

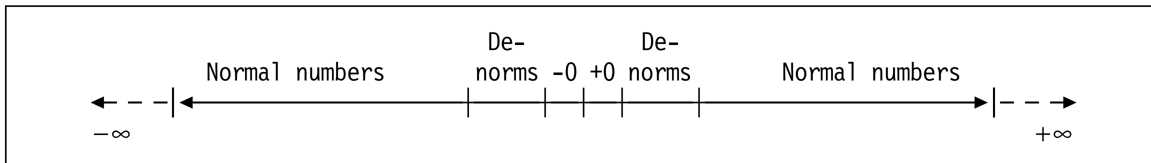


Figure 406. A view of the binary floating-point representation

NaNs are the only representable forms that can't be used in arithmetic operations.

Exercises

34.1.1.(2)+ Write numeric expressions for the values of the three short binary floating-point values

- Min (minimum normal number),
- DMax (maximum denormalized number), and
- DMin (minimum denormalized number).

34.1.2.(2)+ In the short precision binary floating-point representation, how many denormalized numbers can be represented?

34.1.3.(2) Show the hexadecimal representation of +infinity in short, long, and extended precision binary floating-point formats.

34.1.4.(2) Repeat Exercise 34.1.2 for long binary floating-point values.

34.2. Writing Binary Floating-Point Constants

Binary floating-point constants have the same types (E, D, and L) as other floating-point constants, and the type extension must be B.

EB Short format binary floating-point

DB Long format binary floating-point

LB Extended format binary floating-point

You can create short binary floating-point constants like these:

```
SBFP1    DC    EB'1.0'           X'3F800000'  Binary floating-point 1
SBTenth  DC    EB'0.1'           X'3DCCCCCD'  Binary floating-point 1/10
BPi      DC    EB'3.14159265'    X'40490FDB'  Pi
B10Mil   DC    EB'10000000'      X'4B189680'  10**7
B10Milth DC    EB'.0000001'      X'33D6BF95'  10**(-7)
Familiar DC    EB'123.4567'      X'42F6E9D5'  A familiar value
```

Figure 407. Examples of short binary floating-point constants

Converting decimal values to binary floating-point is a bit complicated; if you're interested, here's how it's done for the last constant, +123.4567:

1. The integer part of the constant is 123 = X'7B'.
2. The fraction part, 0.4567, must be converted to binary. Using the method described in Section 31.3 on page 557, the result (written in hexadecimal for compactness) is X'0.74EA4A8C1...'.
3. The two parts combined (again, in hexadecimal) give 123.4567_{10} = X'7B.74EA4A8C1...'.
4. Now, we rewrite this in binary: B'111 1011. 0111 0100 1110 1010 1010 1000...'.
5. We now shift the binary point left by 6 digits, so the value can be rewritten as B'1.11 1011 0111 0100 1110 1010 1010 1000...'$\times 2^6$. The single 1-bit preceding the binary point is because normal binary floating-point values have an implicit high-order 1-bit.

6. We now know the exponent 6; adding the bias (+127, from Table 271 on page 638), the characteristic is 133 = X'85' = B'10000101'.
7. From Table 271 on page 638 the characteristic width is 8 bits; using a zero bit for the sign, we know that the first 9 bits of the constant are 0 1000 0101, or 0100 0010 1.
8. Finally, we attach the fraction bits (omitting the implicit 1-bit): B'0100 0010 1111 0110 1110 1001 1101 0101', or X'42F6E9D5'.
9. The first omitted bit (underscored in step 4 above) is zero, so the result wasn't rounded.

It's easy to see why we leave conversions to the Assembler!

Similarly, you can create long and extended format binary floating-point constants:

```
LBFP1    DC    DB'1.0'            BFP 1, long format
LBTenth  DC    LB'0.1'            BFP 1/10, extended format
LB10Mil  DC    DB'1000000'        BFP 10**7, long format
XBMTenth DC    LB'-.1'            BFP -0.1, extended format
```

Figure 408. Examples of long and extended binary floating-point constants

Rounding is important in binary floating-point arithmetic. You can choose how your constants should be rounded by following the nominal value of a numeric constant with the letter R and a decimal number. The supported rounding indicators are:

R1 Add 1 in the first lost bit position (biased round)

R4 Unbiased round to nearest, ties to even (default)

R5 Round toward zero (truncate, chop)

R6 Round toward maximum positive value (+infinity)

R7 Round toward minimum positive value (−infinity)

If no rounding indicator is specified, the Assembler uses R4, “unbiased round to nearest”. The examples in Figure 409 show how these rounding indicators are used; the first two constants use default rounding.

```
         DC    EB'+0.1'      X'3DCCCCCD'  Unbiased round half-even
         DC    EB'-0.1'      X'BDCCCCCD'  Unbiased round half-even
         DC    EB'+0.1R1'    X'3DCCCCCD'  Biased round
         DC    EB'-0.1R1'    X'BDCCCCCD'  Biased round
         DC    EB'+0.1R4'    X'3DCCCCCD'  Unbiased round half-even
         DC    EB'-0.1R4'    X'BDCCCCCD'  Unbiased round half-even
         DC    EB'+0.1R5'    X'3DCCCCCC'  Truncate
         DC    EB'-0.1R5'    X'BDCCCCCC'  Truncate
         DC    EB'+0.1R6'    X'3DCCCCCD'  Round toward +infinity
         DC    EB'-0.1R6'    X'BDCCCCCC'  Round toward +infinity
         DC    EB'+0.1R7'    X'3DCCCCCC'  Round toward -infinity
         DC    EB'-0.1R7'    X'BDCCCCCD'  Round toward -infinity
```

Figure 409. Rounding indicators for binary floating-point constants

The Assembler also supports nominal-value operands for special values, as shown in Table 275 on page 644.

Operand	Generated value
(Inf)	Infinity
(Max)	Largest magnitude
(Min)	Smallest normalized magnitude
(DMin)	Smallest denormalized magnitude
(SNaN)	A Signaling NaN, with B'01' in the high-order fraction bits and zeros elsewhere
(QNaN)	A Quiet NaN, with B'11' in the high-order fraction bits and zeros elsewhere
(NaN)	A Quiet NaN, with B'10' in the high-order fraction bits and zeros elsewhere

Table 275. Nominal-value operands for binary floating-point special values

These assembled constants are shown for each format in Table 276: in Table 276:

Value	Short	Long	Extended
(Inf)	7F800000	7FF0000000000000	7FFF000000000000 0000000000000000
(Max)	7F7FFFFF	7FEFFFFFFFFFFFFF	7FFEFFFFFFFFFFFF FFFFFFFFFFFFFFFF
(Min)	00800000	0010000000000000	0001000000000000 0000000000000000
(DMin)	00000001	0000000000000001	0000000000000000 0000000000000001
(SNaN)	7FA00000	7FF4000000000000	7FFF400000000000 0000000000000000
(QNaN)	7FE00000	7FFC000000000000	7FFFC00000000000 0000000000000000
(NaN)	7FC00000	7FF8000000000000	7FFF800000000000 0000000000000000

Table 276. Assembled binary floating-point special-value constants

Special values may be signed:

```
         DC    EB'-(Inf)'         X'FF800000'
         DC    DB'-(Min)'         X'8010000000000000
         DC    LB'-(NaN)'         X'FFFF800000000000 0000000000000000'
```

Sometimes it is useful to add information to the unused fraction bits of a NaN, such as its address or the number of the statement in which it was defined. This “payload” can be used for diagnostic information when a numeric result doesn't behave as you might expect.

```
QN       DC    EB'(QNaN)'         Short QNaN
         Org   *-2                Back up location counter by 2
QNParm   DC    H'1324'            Statement 1324, short NaN payload
SN       DC    DB'(SNaN)'         Long SNaN
         Org   *-4                Back up location counter by 4
SNParm   DC    A(SN)              Address of the long SNaN
```

Figure 410. Examples of parameterized binary floating-point NaNs

If you decide to parameterize your NaNs by adding payloads, follow these guidelines:

- Don't use the *two* high-order fraction bits; they are needed to distinguish NaN types.
- If possible, use at most the 21 high-order bits for NaN payload values, in case your NaN might be used in an arithmetic operation that yields a short-format result.

  Thus, in Figure 410 the “**SNParm**” payload might be truncated if the NaN named **SN** is shortened, and the address information would be incorrect.

34.2.1. Decimal Exponents and Exponent Modifiers

As with other numeric floating-point constants, you can specify a decimal exponent with the nominal value, and an exponent modifier to apply to all values in the constant. For example, some of the constants in Figure 407 on page 642 and Figure 408 on page 643 could be written as in Figure 411 on page 645.

```
B10Mil   DC    EB'1E7'            X'4B189680'         Exponent
B10Milth DC    EB'1E-7'           X'33D6BF95'         Exponent

B10Mil   DC    EBE7'1'            X'4B189680'         Modifier
B10Milth DC    EBE-7'1'           X'33D6BF95'         Modifier

LBFP1    DC    DBE-12'1E12'       X'3FF0000000000000' Both
```

Figure 411. Binary floating-point constants with decimal exponents and modifiers

As the constant **LBFP1** shows, the power of ten multiplying the numeric part of the constant is the sum of the exponent modifier (−12) and the decimal exponent (+12), so the generated constant is simply 1.0 in long precision.

34.2.2. Length Modifiers (*)

Length modifiers are rarely used for binary floating-point data. To avoid excessive truncation, the Assembler requires the minimum bit lengths shown in Table 277.

Data Type	Short	Long	Extended
Normal values	9	11	16
Special values	11	14	18

Table 277. Minimum bit lengths for binary floating-point constants

While it may seem strange to allow normal binary floating-point constants to include only the sign and exponent, there is always an *implied* 1-bit preceding the radix point. Thus, you could write

```
         DC    LBL2'1'            X'3FFF'
```

and generate a valid (if not especially useful) constant!

The minimum lengths for special values are two bits larger than for normal values, to make sure there's enough room for the bits distinguishing NaN types.

Advice

Don't use length modifiers with binary floating-point constants.

Binary floating-point constants do not support a Scale modifier.

Exercises

34.2.1.(2) What is the hexadecimal representation of the decimal value 0.1 with lengths 2, 3, and 4 bytes in each of the three formats?

34.2.2.(2)+ A four-byte area of memory contains the bit pattern X'4040405C'. What is represented by that pattern? (You should now be able to describe six different possibilities.)

34.2.3. Determine if each of these short binary floating-point numbers is zero, normal, denormal, infinity, a QNaN, or a SNaN:

1. X'7FFFFFFF'
2. X'007FFFFF'
3. X'80000000'
4. X'00FFFFFF'
5. X'FF8000AB'
6. X'FF800000'

34.2.4.(3) Suppose you define these constants:

```
Short    DC    DBL4'0.1'
ShortOne DC    DBL4'1'

Long     DC    EBL8'0.1'
LongOne  DC    EBL8'1'
```

What values would actually appear when the first pair is used as short operands, and the second pair is used as long operands?

34.3. Binary Floating-Point Arithmetic in General

The IEEE standard[221] specifies a powerful rule:

> Each of the computational operations that return a numeric result ... shall be performed as if it first produced an intermediate result to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit the destination's format.

This rule means that you can know the calculated result in all situations.

34.3.1. Rounding Modes

The standard specifies that "the result cannot suffer more than one rounding error". There are four "global" rounding modes that can be set to apply to subsequent operations; you can change them during your program's execution.

0 Unbiased round to nearest (ties — exact values lying exactly half way between two representations in the target format — operand in FPR0 isn't in long format — it's just the high-order round to the neighbor of the exact value with low-order bit zero)

1 Round toward zero (truncate)

2 Round toward $+\infty$

3 Round toward $-\infty$

Most applications use rounding mode 0 and don't change it during execution.

Some specialized instructions provide an explicit "local" rounding-method *mask field* for the operation of only that one instruction:

0 Round according to the current "global" rounding mode in the Floating-Point Control Register (described in Section 34.4.1)

1 Biased round to nearest (ties round away from zero; effectively, the same as adding 1 to the first lost bit)

4 Unbiased round to nearest

5 Round toward zero (truncate)

6 Round toward $+\infty$

7 Round toward $-\infty$

Most of the conversion instructions between binary floating-point and binary integers provide a "local" rounding mask.

[221] The full name is "IEEE Std 754™-2008".

34.3.2. Denormalized Numbers

Denormalized numbers allow "gradual underflow" if the exponent of a normalized result might fall below E_{min}. This means that binary floating-point arithmetic needs no unnormalizing arithmetic operations.

To illustrate the importance of denormalization, we'll use the notation for FPF(10,4) numbers introduced in Section 32.9 on page 582, where E_{min} is −9. Suppose we have two numbers X = (.1234×10^{-9}) or [−9‖+.1234], and Y = (.1200×10^{-9}), or [−9‖+.1200]. Then X−Y is

```
     -9‖+.1234   X
  -  -9‖+.1200   -Y
  =  -9‖+.0034   intermediate denormalized result
  = -11‖+.3400   normalized (underflowed) result
```

Because the exponent of the normalized result is −11, the result is often set to zero.

Suppose we add Y to (X−Y): we would expect to get X.

```
     -9‖+.1200   Y
  +   0‖+.0000   underflowed difference (X-Y)
  =  -9‖+.1200   normalized result = Y
```

so that *without* denormalization, (X−Y)+Y gives Y! However, if the result of (X−Y) is denormalized, we get

```
     -9‖+.0034   denormalized difference (X-Y)
  +  -9‖+.1200   Y
  =  -9‖+.1234   normalized result = X
```

and (X−Y)+Y gives X, the result we expect.

Denormalized numbers are the default result of gradual underflow.

Of course, normal underflow can occur. In FPF(10,4), if we multiply [−6‖+.2000] by itself, the intermediate result is [−12‖+.0400]. Deormalizing this result while retaining a nonzero fraction can only generate [−10‖+.0004], so that one further denormalization step to increase the exponent to −9 generates a zero fraction. This is a true underflow.

It sometimes helps to visualize a line of binary floating-point values close to zero. If values smaller than E_{min} are normalized (or "flushed" to zero), there are no represented values between E_{min} and zero. However, with gradual underflow there are as many values between between E_{min} and zero as between between E_{min} and between E_{min}+1. In Figure 412 (a "magnification" of the region near zero in Figure 406 on page 642), we suppose the binary floating-point numbers have 3 fraction bits:

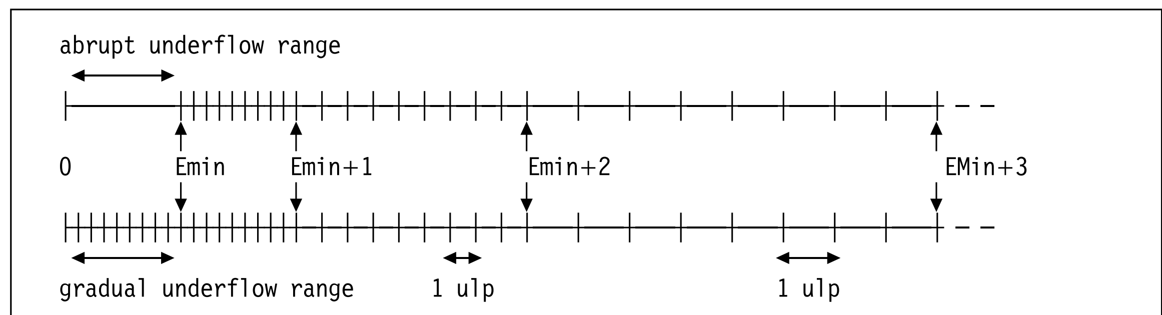


Figure 412. Values representable with gradual underflow

As noted in Section 32.8 on page 580, an "ulp" of a number varies with the magnitude of the number. As the figure illustrates, an ulp is proportional to the spacing between neighboring representable values.

34.3.3. Arithmetic with Zero, Infinity, and NaNs

The IEEE standard defines actions taken when operations involve zero, infinity, or NaNs.

1. **Zero**

   In operations with finite nonzero values Fn, zero behaves as you would expect: 0×Fn is zero, 0±Fn and Fn±0 all return that finite number Fn.[222]

   Some situations require creation of −0, such as:

   - +0×(negative finite number), or −0×(positive finite number)
   - +0÷(negative finite number), or −0÷(positive finite number)
   - (Fn−Fn), rounded toward −∞
   - (−0)+(−0)
   - Square root of −0

2. **Infinity**

   Valid arithmetic on infinities is always “exact”, and correctly signed. If Fn is a finite nonzero number, then:

   - Fn ± ∞ = ±∞
   - |∞|+|∞| = ∞
   - 0 × ∞ = invalid-operation exception; if the exception is masked off, a default NaN is returned
   - Fn × ∞ = ∞
   - Fn ÷ ∞ = 0
   - Fn ÷ 0 = ∞
   - 0 ÷ 0 = invalid-operation exception; if the exception is masked off, a default NaN is returned

   and other similar operations. As noted, some operand combinations result in exceptions, as discussed in Section 34.4.

   Note, however, that infinity doesn't behave like a finite number, where (1÷X)×X=1, X÷X=1, and X−X=0. If X=∞, none of these relations apply!

3. **NaNs**

   NaN operands in arithmetic operations have several possible results:

   - A SNaN will either cause an invalid-operation exception or produce a QNaN as the result, depending on the setting of a mask bit (described shortly, in Section 34.4).
   - QNaNs are propagated without causing an exception; if both operands are QNaNs, the CPU will deliver operand 1.

Exercises

34.3.1.(3) Show in hexadecimal the result generated if you calculate `(Min)-(DMin)` in short binary floating-point.

Details

The next section describes binary floating-point exceptions. If you are interested mainly in the arithmetic operations, skip ahead to Section 34.5.

[222] A Cornell student once asked a professor, “Is zero a number?” He replied, “Oh yes! It's one of the best!”

34.4. Binary Floating-Point Exceptions, Interruptions, and Controls

The IEEE binary floating-point standard also defines a rich set of controls for exceptions, when a valid result cannot be generated. Unlike hexadecimal floating-point, where only exponent underflow and lost significance can be controlled by bits in the Program Mask, binary floating-point lets you control *five* exception conditions.

1. **Invalid operation** has two causes:
   - Mathematically meaningless operations, such as
     a. $|\infty| - |\infty|$
     b. $\infty \times 0$
     c. $\infty \div \infty$
     d. $0 \div 0$
     e. Remainder of $X \div Y$ with $X = \infty$ or $Y = 0$
     f. Square root of a negative value
   - Computationally meaningless operations, such as
     a. an operation on a Signaling NaN (SNaN)
     b. a comparison operation involving a NaN (an “unordered” comparison)
     c. an integer-conversion fault (where the source operand is a NaN, ∞, or is too large for the target operand)
2. **Division by zero** ($0 \div 0$ is an invalid operation).
3. **Exponent overflow**: the magnitude of the result is too large for the target format.
4. **Exponent underflow**: the magnitude of the result is too small for the target format.
5. **Inexact result** occurs when a rounding operation changes the value of the exact infinite-precision unbounded-range result. This condition can occur with others.

34.4.1. Binary Floating-Point Exceptions (*)

Exceptions can cause an interruption (the IEEE standard calls it a “trap”), or set a status flag bit. Which of these two actions occurs depends on the setting of a *mask* bit.

The mask bits and the status flags are held in the Floating-Point Control (FPC) Register; it is distinct from all other registers. In addition to the mask and status flag bits, the FPC register contains a Data Exception Code (DXC) and bits for the “global” rounding mode. Figure 413 illustrates the format of the FPCR.

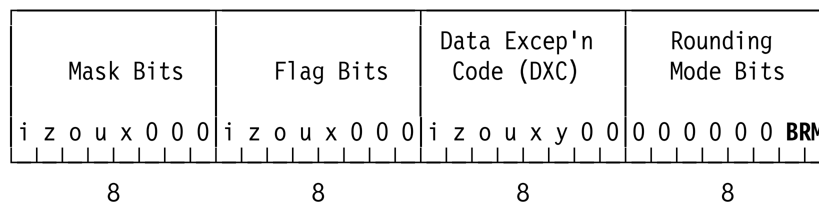


Figure 413. Floating-Point Control (FPC) register

- The mask and flag bits in the first and second bytes are indicated by single letters:

  Bit **Meaning**

  i (bit 0) Invalid operation

  z (bit 1) Divide by zero

  o (bit 2) Overflow

  u (bit 3) Underflow

  x (bit 4) Inexact result

  When an exception occurs, the CPU takes one of two actions:

- If the corresponding mask bit is one an interruption occurs, the Interruption Code is set to 7, and information about the cause is placed in the Data Exception Code (DXC) byte and in memory at address 147 (X'93'). The corresponding flag bit is unchanged.
- If the corresponding mask bit is zero a default action is taken, no interruption occurs, its corresponding status flag bit is set to 1, and the DXC remains unchanged. A default result value is generated.

  Once a status flag bit is set to 1, it is unchanged until it is explicitly reset by an instruction that changes the contents of the FPC register.

Sometimes the mask bits in the first byte are called “Interruption Mask” bits, abbreviated IMi, IMz, etc.; and the flag bits in the second byte are called “Status Flag” bits, abbreviated SFi, SFz, etc.

- The third byte of the FPC register holds the DXC; its value is set when an interruption occurs. The **y** in Figure 413 on page 649 appears only in the DXC:

  Bit **Meaning**

  y (bit 5) The result was incremented

  Note that the first five bits of the DXC shown in Figure 413 on page 649 correspond from left to right to the first five bits of the mask and flag bits.

  If an interruption occurs the Interruption Code is set to 7, the same IC used for the (packed decimal) Data Exception. To learn whether the interruption is due to a binary floating-point or packed decimal operation, you must test the DXC for zero (packed decimal) or nonzero (binary floating-point).

  Table 278 describes the values placed in the DXC:

DXC	Meaning
00	Decimal data exception (for invalid packed decimal data, as discussed in Section 29)
01	An invalid Floating-Point Register is used by a floating-point instruction; the registers are available on this CPU, but are not enabled
02	a BFP instruction is attempting to execute on a CPU where BFP instructions are available, but are not enabled
08	IEEE inexact exception; the result was truncated
0C	IEEE inexact exception; the result was incremented
10	IEEE underflow exception; the result is exact
18	IEEE underflow exception; the result is inexact and was truncated
1C	IEEE underflow exception; the result is inexact and was incremented
20	IEEE overflow exception; the result is exact
28	IEEE overflow exception; the result is inexact and was truncated
2C	IEEE overflow exception; the result is inexact and was incremented
40	IEEE division by zero
80	IEEE invalid operation

Table 278. Binary floating-point DXC values

- The binary floating-point rounding mode is specified by the two low-order bits of the fourth byte of the FPC register, indicated by **BRM** in Figure 413 on page 649. They have the same values we saw for the rounding modes discussed in Section 34.3.1 on page 646:

BRM	Meaning
B'00'	To nearest
B'01'	Toward zero (truncate)
B'10'	Up (toward $+\infty$)
B'11'	Down (toward $-\infty$)

Comment

When an exception condition occurs, we may say that it is “signaled” or “indicated” or “raised”; each term means the same. The setting of the appropriate mask bit in the FPCR determines subsequent actions.

34.4.2. FPC Register Instructions (*)

The five instructions in Table 279 are used to manage the FPC register.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B29D	LFPC	S	Load FPC	B2BD	LFAS	S	Load FPC and Signal
B384	SFPC	RRE	Set FPC	B385	SFASR	RRE	Set FPC and Signal
B38C	EFPC	RRE	Extract FPC	B29C	STFPC	S	Store FPC
B299	SRNM	S	Set BFP Rounding Mode				

Table 279. Binary floating-point FPC register control instructions

The Load FPC and Store FPC instructions use a 32-bit operand in memory. They are used to set the mask and Rounding Mode bits. For example:

```
STFPC OldFPC            Store current FPCR into memory
LFPC  NewFPC            Load new FPCR from memory
```

The flag bits and the Data Exception Code are set by the CPU when an exception condition occurs.

The Set FPC and Extract FPC instructions use a 32-bit operand in the rightmost 32 bits of a general register; bits 0-31 in the left half of the register are unaffected. For example:

```
SFPC  1                 Set FPCR from GPR 1
EFPC  1                 Extract FPCR into GPR 1
```

The SRNM (Set BFP Rounding Mode) instruction takes its operand from the rightmost 2 bits of its Effective Address and puts them in the “BRM” (Binary Rounding Mode) bits of the FPCR. The rest of the FPCR is unchanged. For example:

```
SRNM  0(5)              Set BRM from last 2 bits of GPR5
SRNM  1                 Set rounding mode to 'toward zero'
```

The LFAS and SFASR instructions load and set the FPCR, but with the side effect of possibly generating a specific exception condition. They are used to simulate BFP exception conditions without needing to perform actual BPF operations that would generate the exceptions. They make it easier to write and test exception handling routines. Because their use is specialized, we won't discuss them further.

34.4.3. Exception Actions (*)

When an exception occurs, the generated result depends on whether an interruption occurs (the corresponding mask bit is 1) or does not (the corresponding mask bit is 0).

1. **Invalid operation**

Mask bit = 1	**Mask bit = 0**
An interruption occurs: the DXC is set to X'80', and the instruction is suppressed (registers remain unchanged). The **i** flag bit is unchanged.	A special "hardware generated" QNaN is delivered as the default result: its first fraction bit is 1, and the rest of the fraction is zero. If an operand was a SNaN, the corresponding QNaN is delivered. The **i** flag bit is set to 1.

Table 280. Invalid operation binary floating-point exception

If the two operands are QNaN *and* SNaN, the SNaN takes precedence as the result, even if it is forced to a QNaN.

2. **Divide by zero**

Mask bit = 1	**Mask bit = 0**
An interruption occurs: the DXC is set to X'40', and the instruction is suppressed (registers remain unchanged). The **z** flag bit is unchanged.	A ±infinity is delivered. The **z** flag bit is set to 1.

Table 281. Divide by zero binary floating-point exception

3. **Overflow**

Mask bit = 1	**Mask bit = 0**
An interruption occurs: the DXC is set to X'20', X'28', or X'2C', and the exponent of the result is "scaled". The **o** flag bit is unchanged.	A correctly signed infinity or MaxReal is delivered, depending on the current rounding mode. The **o** and **x** flag bits are set to 1.

Table 282. Exponent overflow binary floating-point exception

Be Careful!

If possible, don't set overflowed results to MaxReal. MaxReal is a valid finite number, so it might propagate unnoticed through subsequent computations.

4. **Underflow**

Mask bit = 1	**Mask bit = 0**
An interruption occurs: the DXC is set to X'10', X'18', or X'1C', and the exponent of the result is "scaled". The **u** flag bit is unchanged.	A zero or denormalized result is delivered. The **u** flag bit is set to 1, and the **x** flag bit is set to 1 if the result is inexact.

Table 283. Exponent underflow binary floating-point exception

5. **Inexact result** (An inexact exception can also occur with BFP overflow or underflow.)

Mask bit = 1	**Mask bit = 0**
An interruption occurs: the DXC is set to X'08' or X'0C'. The **x** flag bit is unchanged.	The result is delivered. The **x** flag bit is set to 1.

Table 284. Inexact result binary floating-point exception

The mask and flag bits in the FPC register are ignored for HFP and FP Support instructions.

Remember

Binary floating-point exceptions may or may not cause a program interruption, depending on the settings of mask bits in the FPC register, and the generated results can be quite different.

34.4.4. Scaled Exponents (*)

When an exponent overflow or underflow is not masked the CPU generates a program interruption, and the characteristic of the result is adjusted or “scaled” by adding (for underflows) or subtracting (for overflows) a fixed quantity. This adjustment brings the scaled result near the middle of the characteristic range, so that the decimal exponent is nearer to zero than the ends of the exponent range. The scaling factors are shown in Table 285.

Data length	Scale Factor
Short	192
Long	1536
Extended	24576

Table 285. BFP overflow/underflow scale factors

To see how this works, consider multiplying two short binary floating-point numbers that cause overflow. If their exponents are both E_{max} = +127, the exponent of the product will be about +254. By subtracting the scale factor (192), the exponent of the product will be about +62, nearly half of E_{max}, and easier to manage if further adjustments are needed.

Exercises

34.4.1.(1)+ What binary floating-point exception conditions and rounding mode will be set by this instruction?

```
LFPC  =X'30000003'
```

34.4.2.(2) What will be the effect of executing these three instructions?

```
EFPC  9
NILH  9,=X'7F00'
SFPC  9
```

34.4.3.(2)+ In Table 278 on page 650, a Data Exception Code X'02' means that you have tried to execute a BFP instruction on a CPU where BFP instructions are available, but not enabled. What do you think will happen if you try to execute a BFP instruction on a CPU where the BFP instructions are not available?

34.4.4.(1)+ Refer to Table 278 on page 650 and construct a table that identifies the meaning of *each* of the 8 bits in the DXC.

34.5. Basic Binary Floating-Point Instructions

Before we discuss the basic binary floating-point instructions, it's worth reviewing the instructions in Section 31.9 starting on page 568. Those instructions can move floating-point operands of all representations, so they can be used with hexadecimal floating-point, binary floating-point, and decimal floating-point data.

Some general points to remember about binary floating-point instructions:

- Almost all binary floating-point instruction mnemonics contain a letter 'B'.
- Their actions are more complex than the equivalent operations in hexadecimal floating-point, due to the presence of rounding modes, NaNs and infinities, maskable exceptions, and different exponent ranges.

- Instructions involving extended-precision operands must specify a valid pair of floating-point registers.
- They support a full range of common algebraic operations.

Unlike hexadecimal floating-point, binary floating-point supports non-numeric values. Because it's difficult to test an operand for a special value by examining its bit patterns, we use the “Test Data Class” instructions listed in Table 286.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED10	TCEB	RXE	Test Data Class (Short)	ED11	TCDB	RXE	Test Data Class (Long)
ED12	TCXB	RXE	Test Data Class (Extended)				

Table 286. Binary floating-point Test Data Class instructions

A bit pattern in the rightmost 12 bits of the Effective Address of the second operand address tests selected classes of the first operand. The test bits are shown in Table 287.

Class	+ sign	− sign
Zero	52	53
Norm	54	55
Denorm	56	57
Infinity	58	59
QNaN	60	61
SNaN	62	63

Table 287. Test Data Class second-operand bits

Another view of the corresponding positions of the bits in the second operand Effective Address and the tested value is illustrated in Table 288.

Bits 0-51	52	53	54	55	56	57	58	59	60	61	62	63
Ignored	+Zero	−Zero	+Normal	−Normal	+Denormal	−Denormal	+Infinity	−Infinity	+QNaN	−QNaN	+SNaN	−SNaN

Table 288. Test Data Class second-operand test-bit/tested-value correspondence

If any test bit corresponds to the class of the first operand, the Condition Code is set to 1; otherwise, it is set to 0. You can use these instructions to test for SNaNs without causing an invalid-operation Exception.

The 12 bits of the test pattern fit in the instruction's displacement field, so you can specify it both at assembly time and at execution time. For example, suppose you want to test a short binary floating-point operand for ±infinity:

```
        LE    2,FPData          Get the operand
        TCEB  2,X'030'          Test for + or - infinity
        JNZ   AnInfin           Branch if yes
or
        LA    9,B'110000'       Put test mask bits in GR9
        TCEB  2,0(,9)           Same test
        JNZ   AnInfin           Branch if yes
```

The TDEB instructions use the same test bit pattern.

Exercises

34.5.1.(2) Given a binary floating-point operand in FPR0, which data classes will these instructions detect?

```
(1)      TCEB  0,29
(2)      TCDB  0,4095
(3)      TCXB  0,15
(4)      TCDB  0,X'C00'
```

34.5.2.(3) What result would you expect from executing each pair of these instructions:

```
(1)      LE    2,=DB'(Inf)'
         TCEB  2,X'30'          Test for infinity
(2)      LE    4,=EB'42'
         TCXB  4,X'300'         Test for normal
(3)      LE    6,=EB'42'
         TCDB  6,X'300'         Test for normal
```

34.5.3.(1)+ Which instruction in Exercise 34.5.1 is redundant?

34.5.4.(3) For each data item in Exercise 34.2.3, what operand of the TCxB instruction should be used to correctly determine the class of the item?

34.6. Binary Floating-Point RR-Type Data Movement Instructions

Remember that the instructions described in Section 31.9 can be used to move any floating-point data between registers and memory. The instructions in Table 289 differ from the related representation-independent instructions because they are sensitive to the presence of NaN operands.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B302	LTEBR	RRE	Load and Test (Short)	B303	LCEBR	RRE	Load Complement (Short)
B301	LNEBR	RRE	Load Negative (Short)	B300	LPEBR	RRE	Load Positive (Short)
B312	LTDBR	RRE	Load and Test (Long)	B313	LCDBR	RRE	Load Complement (Long)
B311	LNDBR	RRE	Load Negative (Long)	B310	LPDBR	RRE	Load Positive (Long)
B342	LTXBR	RRE	Load and Test (Extended)	B343	LCXBR	RRE	Load Complement (Extended)
B341	LNXBR	RRE	Load Negative (Extended)	B340	LPXBR	RRE	Load Positive (Extended)

Table 289. Binary floating-point RR-type data movement instructions

For the operations Load Complement, Load Negative, and Load Positive, the result placed at the first operand will be unmodified except for a possible sign change, even if the operand is a NaN. For short operands, the right half of the target register is unchanged.

The CC setting is shown in Table 290 on page 656:

CC	Meaning
0	Operand is zero
1	Operand is less than zero
2	Operand is greater than zero
3	Operand is a NaN

Table 290. CC settings for BFP data movement instructions

The Load and Test instructions are sensitive to the difference between a QNaN and an SNaN. If the second operand is a Signaling NaN, an invalid operation exception occurs, and the result depends on the setting of the invalid-operation mask bit in the FPCR:

- If the mask bit in the FPCR for an invalid operation is 1, an interruption occurs, and the Condition Code is unchanged.
- If the mask bit is 0, no interruption occurs, and the result is the corresponding QNaN.
- If no interruption occurs, the Condition Code is set by the Load and Test instructions as shown in Table 290.

Figure 414 shows some examples of these instructions:

```
         LE    0,=EB'(QNaN)'      c(FPR0)=X'7FE00000'
         LCEBR 2,0                c(FPR2)=X'FFE00000', CC=1
         LTEBR 4,0                c(FPR4)=X'7FE00000', CC=3
         LNEBR 6,0                c(FPR6)=X'FFE00000', CC=1
*
         LD    0,=DB'-3.14'       c(FPR0)=X'C0091EB851EB851F'
         LTDBR 0,0                CC=1
         LPDBR 2,0                c(FPR2)=X'40091EB851EB851F', CC=2
```

Figure 414. Examples of binary floating-point data movement instructions

Don't mix floating-point data types

You might accidentally use a hexadecimal floating-point operation on binary floating-point data, or vice versa. The results may be unexpected!

Exercises

34.6.1.(1) If you execute these instructions:

```
LD    4,=X'50123456789ABCDE'
LD    6,=X'FEDCBA9876543210'
LTXBR 0,4
```

What will be in the register pair FPR(0,2), and what will be the CC setting?

34.6.2.(2) Suppose c(FPR0)=X'0A123456789ABCDE' and c(FPR2)=X'42857196DBB93310'. Show the CC setting and the contents of the result register or registers after executing each of the following instructions:

```
(1)        LPEBR 4,2
(2)        LTDBR 2,2
(3)        LCXBR 4,0
(4)        LCDBR 4,2
```

34.6.3.(2) Suppose you execute these sets of instructions. What will be the resulting CC setting for each case?

```
(a)      LE    0,=EB'(SNaN)'
         LTER  0,0

(b)      LE    2,=X'FFFFFFFF'
         LCEBR 2,2
```

34.6.4. Which instruction in Exercise 34.5.1 could be replaced by one other instruction, and what is that instruction?

34.7. Binary Floating-Point Multiplication

Table 291 lists the instructions for binary floating-point multiplication. None of them change the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED17	MEEB	RXE	Multiply (Short←Short×Short)	B317	MEEBR	RRE	Multiply (Short←Short×Short)
ED0C	MDEB	RXE	Multiply (Long←Short×Short)	B30C	MDEBR	RRE	Multiply (Long←Short×Short)
ED1C	MDB	RXE	Multiply (Long←Long×Long)	B31C	MDBR	RRE	Multiply (Long←Long×Long)
ED07	MXDB	RXE	Multiply (Extended←Long×Long)	B307	MXDBR	RRE	Multiply (Extended←Long×Long)
				B34C	MXBR	RRE	Multiply (Ext.←Ext.×Ext.)

Table 291. Binary floating-point Multiply instructions

As the table indicates, five instructions (MEEB, MEEBR, MDB, MDBR, and MXBR) generate a product the same length as the two operands, while four others (MDEB, MDEBR, MXDB, and MXDBR) generate a double-length product. As with other multiply instructions, the sign of the product is the XOR of the signs of the two operands, even if they are zero or infinity. MXBR, like other instructions handling extended-precision operands, must refer to valid floating-point register pairs.

Multiplying two finite values gives the expected rounded result, determined by the rounding mode in the FPCR, so long as there is no overflow or underflow. Figure 415 gives an example, where we've assumed the default rounding mode (to nearest).

```
         LE    0,=EB'0.1'        c(FPR0)=X'3DCCCCCD'
         MDEB  0,=EB'-0.2'       c(FPR0)=X'BF947AE151EB8520' Long product

         LE    0,=EB'0.1'        c(FPR0)=X'3DCCCCCD'
*                                Multiply by X'BE4CCCCD' (= -0.2)
         MEEB  0,=EB'-0.2'       c(FPR0)=X'BCA3D70B'        Short product
```

Figure 415. Example of binary floating-point multiply instructions

The first multiplication (using MDEB) creates a double-length product of the two operands; its fraction digits start with X'47AE151E...'. The single-length product (using MEEB) has a shorter exponent field, so its fraction digits are X'47AE16'. Because the longer result is slightly closer to X'47AE16' than to X'47AE14', default rounding generates the indicated result.

But: the short product is not exact. If the inexact ("x") mask bit in the FPCR is 1, a program interruption will occur. Because we normally don't care about unused trailing bits of a product, that mask bit is often set to zero to suppress the inexact exception.

Figure 416 on page 658 shows how overflow and underflow can occur when we multiply two sufficiently large or sufficiently small numbers:

```
* (1) With the overflow mask bit set to 1:
         LFPC  =X'F8000000'       All mask bits = 1 (allow interrupts)
         LE    0,=EB'1E20'        c(FPR0)=X'60AD78EC'
         MEEBR 0,0                c(FPR0)=X'21EB1950' Overflowed result
         LE    0,=EB'1E-30'       c(FPR0)=X'0DA24260'
         MEEBR 0,0                c(FPR0)=X'3BCDB025' Underflowed result

* (2) With the overflow and underflow mask bits set to 0:
         LFPC  =F'0'              All mask bits = 0 (no interrupts)
         LE    0,=EB'1E20'        c(FPR0)=X'60AD78EC'
         MEEBR 0,=EB'-(Inf)'      c(FPR0)=X'7F800000' Overflow to +infinity
         LE    0,=EB'1E-30'       c(FPR0)=X'0DA24260'
         MEEBR 0,0                c(FPR0)=X'00000000' Underflow to zero
```

Figure 416. Examples of binary floating-point multiplication overflow and underflow

In case (1), both MEEBR instructions cause interruptions, and the result in FPR0 is set to a scaled result: the significand is correct, but the exponent has been adjusted by a fixed amount to bring the result into a more manageable range. Further computation with this result must take account of the exponent scaling.

In case (2) both interruptions are suppressed, and the default result is +infinity for overflow, and zero for underflow.

Figure 417 shows two examples with signed zeros and infinities:

```
         LE    0,=EB'-0.'        c(FPR0)=X'80000000' (-0)
         MEEBR 0,0               c(FPR0)=X'00000000' (-0)*(-0) = +0

         LE    0,=EB'(Inf)'      c(FPR0)=X'7F800000' (+infinity)
         MEEBR 0,=EB'-(Inf)'     c(FPR0)=X'FF800000' (-infinity)
```

Figure 417. Examples of binary floating-point multiply instructions

Figure 418 shows how a denormalized result can be generated:

```
         LFPC  =F'0'              Set all mask bits to zero
         LE    0,=EB'1E-21'       c(FPR0)=X'1C971DA0'
         MEEBR 0,0                c(FPR0)=X'000002CA'
```

Figure 418. Example of binary floating-point denormalized product

If the mask bit for exponent underflow had not been set to zero, an underflow exception would have occurred, and the result would have a “wrapped” exponent rather than a denormalized number.

Table 291 on page 657 shows that the product of two extended operands is formed by the RRE-type instruction MXBR, so that both operands must be loaded into register pairs, as illustrated in Figure 419:

```
         LD    0,=LB'0.1'         c(FPR0)=X'3FFB999999999999'
         LD    2,=LB'0.1'+8       c(FPR2)=X'999999999999999A'
         LD    4,=LB'-0.2'        c(FPR4)=X'BFFC999999999999'
         LD    6,=LB'-0.2'+8      c(FPR6)=X'999999999999999A'
         MXBR  0,4
*        Now, c(FPR0,FPR2)=X'BFF947AE147AE147 AE147AE147AE147C'
```

Figure 419. Example of binary floating-point extended-precision operands

The product contains a repeating digit pattern X'147AE'; the final digit has been rounded up to the nearest even value. As in Figure 415 on page 657, if the mask bit for inexact result had been one, an interruption for an inexact exception would have occurred.

If the operands are not finite numbers, some special rules apply:

- Multiplying two infinities, or a finite nonzero number by infinity, generates an infinity.
- Multiplying zero and infinity generates an invalid-operation exception. If the corresponding mask bit is zero, the generated result is a default QNaN.
- If both operands are QNaNs, the first operand is generated.
- If either operand is a SNaN, an invalid-operation exception is generated. If the corresponding mask bit is zero, the generated result is the corresponding QNaN. (If both operands are SNaNs, the first operand is used to create the corresponding QNaN.)

As these rules indicate, the result depends on the order of the operands only if one or both is a NaN. For other values, the order of the operands doesn't matter.

Exercises

34.7.1.(2) Show the actual bit patterns of the two products in Figure 415 on page 657 and determine how the rounded product of the second multiplication is formed.

34.7.2.(2) Write a program segment using binary floating-point arithmetic that will compute a table of the cubes of the first 100 integers, and store them as short binary floating-point numbers starting at **BCubes**.

34.7.3.(3)+ How can you generate the maximum positive short-precision binary floating-point number by multiplying two finite short binary floating-point values?

34.7.4.(2) In Figure 416 on page 658 part (1), determine the exponent of each operand before executing the instruction, and the scaled exponent of each result.

34.8. Binary Floating-Point Division

Table 292 lists the five binary floating-point divide instructions. None of them generate a remainder; the instructions described in Section 34.13 on page 668 can be used to calculate a remainder in most cases. The Condition Code is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED0D	DEB	RXE	Divide (Short)	B30D	DEBR	RRE	Divide (Short)
ED1D	DDB	RXE	Divide (Long)	B31D	DDBR	RRE	Divide (Long)
				B34D	DXBR	RRE	Divide (Extended)

Table 292. Binary floating-point Divide instructions

As this table shows, none of the instructions operate on mixed-length operands: dividend, divisor, and quotient all have the same length. DXBR, like other instructions handling extended-precision operands, must refer to valid floating-point register pairs.

If the result is finite, zero, or infinity (and not a NaN), the sign of the quotient is the XOR of the operand signs.

Figure 420 shows two examples of binary floating-point division.

```
LE    2,=EB'7'
DEB   2,=EB'2'          Result = 3.5 = X'40600000'

LD    0,=DB'987654321'
DDB   0,=DB'3'          Result = 329218107 = X'41B39F783B000000'
```

Figure 420. Examples of binary floating-point division

The quotient is rounded according to the current rounding mode in the Floating-Point Control Register.

Division can also generate overflow and underflow exceptions, as illustrated in Figure 421:

```
* (1) With the overflow mask bit set to 1:
        LFPC  =X'F8000000'      All mask bits = 1 (allow interrupts)
        LE    0,=EB'1E20'       c(FPR0)=X'60AD78EC'
        LER   4,0               Copy to FPR4
        LE    2,=EB'1E-30'      c(FPR2)=X'0DA24260'
        DEBR  0,2               c(FPR0)=X'3288D876' Overflowed result
        DEBR  2,4               c(FPR0)=X'4C6F73D2' Underflowed result

* (2) With the overflow and underflow mask bits set to 0:
        LFPC  =F'0'             All mask bits = 0 (no interrupts)
        LE    0,=EB'1E20'       c(FPR0)=X'60AD78EC'
        LER   4,0               Copy to FPR4
        LE    2,=EB'1E-30'      c(FPR2)=X'0DA24260'
        DEBR  0,2               c(FPR0)=X'7F800000' Overflow to +infinity
        DEBR  2,4               c(FPR0)=X'00000000' Underflow to zero
```

Figure 421. Examples of binary floating-point division overflow and underflow

In case (1), the overflow and underflow exceptions cause an interruption, and the result in the first-operand register has the correct significand and a scaled exponent. In case (2), no interruption occurs, and the default result is delivered (as indicated in the comment fields of the DEBR instructions).

Special cases are treated as follows:

- A finite number divided by infinity returns a correctly signed zero.
- A finite number divided by zero causes a divide-by-zero exception. If the “z” mask bit is zero, a properly signed infinity is generated. For example:

```
        LFPC  =F'0'             Mask off all exceptions
        LE    2,=EB'-3'         Finite dividend to divide by zero
        DEB   2,=EB'+0'         Result = X'FF800000' = -infinity
```

- $0 \div 0$ and $\infty \div \infty$ cause an invalid-operation exception. If masked off, they deliver the default QNaN of the appropriate length.
- If neither operand is a SNaN, but one or both is a QNaN, then QNaN is the generated result.
- An invalid-operation exception results if either operand is a SNaN. If the exception is masked off, the corresponding QNaN is the delivered result.

Exercises

34.8.1.(3) How can you generate the maximum positive short-precision binary floating-point number by dividing two finite short binary floating-point values?

34.8.2.(1) Show the short binary floating-point result of each of these division operations:

1. $+1 \div -1$
2. $-0 \div +\infty$
3. $+\infty \div -1$

34.8.3.(1)+ What values will result from these operations?

1. $+2.4 \div -\infty$
2. $-2.5 \div -\infty$
3. $+2.6 \div -2.6$

34.8.4.(3) Using binary floating-point divide instructions (no “Test Data Class”), how can you distinguish between +0 and −0?

34.9. Binary Floating-Point Addition and Subtraction

Table 293 lists the instructions for binary floating-point addition and subtraction. As with the multiplication and division instructions, extended operands use only RR-type instructions that require valid floating-point register pairs.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED0A	AEB	RXE	Add (Short)	B30A	AEBR	RRE	Add (Short)
ED1A	ADB	RXE	Add (Long)	B31A	ADBR	RRE	Add (Long)
				B34A	AXBR	RRE	Add (Extended)
ED0B	SEB	RXE	Subtract (Short)	B30B	SEBR	RRE	Subtract (Short)
ED1B	SDB	RXE	Subtract (Long)	B31B	SDBR	RRE	Subtract (Long)
				B34B	SXBR	RRE	Subtract (Extended)

Table 293. Binary floating-point Add and Subtract instructions

The machine instruction statement format for these instructions is written

```
mnemonic  R1,R2          RRE format instructions
mnemonic  R1,D2(X2,B2)   RXE format instructions
```

The second operand is added to or subtracted from the first operand, and the resulting sum or difference replaces the first operand.

Table 294 shows the Condition Code settings:

CC	Meaning
0	Result is zero
1	Result is < zero
2	Result is > zero
3	Result is a NaN

Table 294. CC settings after BFP add/subtract instructions

For example:

```
LE    2,=EB'2.55'
AEB   2,=EB'-2.77'       CC=1, result < 0
```

These considerations apply to the instructions:

1. The current rounding mode in the Floating-Point Control Register is used.
2. Denormalized operands are valid, and the result may be denormalized.
3. If both operands are QNaNs, the result is the first operand; and if both operands are SNaNs and the invalid-operation exception is masked off, the result is the QNaN derived from the first operand.
4. There are no instructions for adding or subtracting operands of different lengths. If you need to use mixed-length operands, you must use one of the rounding instructions (to make an operand shorter) or lengthening instructions (to make an operand longer) described in Section 34.11 on page 664.

Figure 422 shows examples of binary floating-point addition and subtraction:

```
LD    0,=DB'1.1E4'       c(FPR0)=X'40C57C0000000000'
ADBR  0,0                c(FPR0)=X'40D57C0000000000', CC=2
LD    2,=DB'1.9E4'       c(FPR2)=X'40D28E0000000000'
SDBR  2,0                c(FPR2)=X'C0A7700000000000', CC=1
```

Figure 422. Examples of binary floating-point addition and subtraction

Because the ADBR instruction in Figure 422 doubles the number in FPR0, its significand is unchanged and its exponent is increased by 1.

Exercises

34.9.1.(1)+ In each binary floating-point representation, show the hexadecimal value of (DMin)+(DMin). Assume that all exceptions are masked off.

34.9.2.(2)+ In each binary floating-point representation, show the hexadecimal value of (Min)+(Min). Do not assume that all exceptions are masked off.

34.9.3.(1)+ What is the result of (1) $(+\infty)-(-\infty)$, (2) $(+\infty)-(+\infty)$?

34.10. Binary Floating-Point Comparison

Table 295 lists the “normal” BFP compare instructions (the “special” instructions are in Section 34.10.1):

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED09	CEB	RXE	Compare (Short)	B309	CEBR	RRE	Compare (Short)
ED19	CDB	RXE	Compare (Long)	B319	CDBR	RRE	Compare (Long)
				B349	CXBR	RRE	Compare (Extended)

Table 295. Binary floating-point Compare instructions

For instructions dealing with extended-precision operands, the operands must be loaded into the floating-point registers.

None of the comparison instructions compare mixed-length operands, but the lengthening instructions described in Section 34.11.2 on page 665 can be used to extend a shorter operand to the length of the longer before comparing.

The Condition Code settings for the compare instructions are shown in Table 295:

CC	Meaning
0	Operand1 = Operand2
1	Operand1 < Operand2
2	Operand1 > Operand2
3	Operands are unordered

Table 296. CC settings for BFP comparisons

All values other than NaNs can be compared:

- +infinity is greater than all values, other than +infinity (to which it compares equal).
- −infinity is less than all values, other than −infinity (to which it compares equal).
- +0 and −0 compare equal. (They are the only two distinct binary floating-point bit patterns for numeric values that compare equal.)

When one or both operands is a QNaN, the comparison is “unordered” and the CC is set to 3. If either operand is a SNaN, an invalid-operation exception either causes an interruption (if the corresponding mask bit is 1), or sets the CC to 3 (if the mask bit is 0).

Figure 423 on page 663 shows examples of binary floating-point comparisons.

```
LE    0,=EB'+0'         c(FPR0) = +0
CEB   0,=EB'-(Inf)'     CC=2 (0 > -infinity)
CEB   0,=EB'-0'         CC=0 (+0 = -0)
CEB   0,=EB'(QNaN)'     CC=3 (unordered)
CEB   0,=EB'+(Inf)'     CC=1 (0 < +infinity)

CDBR  6,11              c(FPR6) : c(FPR11) (long operands)
CDB   4,=DB'5.75E4'     c(FPR6) : 5.75E4 (long operands)
CXBR  0,4               c(FPR0,FPR2) : c(FPR4,FPR6) (ext.)
```

Figure 423. Examples of binary floating-point comparison

Remember that Condition Code 3 does not occur for hexadecimal floating-point comparisons.

34.10.1. Compare and Signal (*)

Specialized uses

These instructions are used mainly for testing binary floating-point applications.

The binary floating-point compare and signal instructions in Table 297 behave just like the "normal" binary floating-point comparisons in Table 295 on page 662, except that *any* NaN causes an invalid-operation exception. Thus, these compare and signal instructions are very useful in helping to prevent propagation of invalid results throughout a calculation.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED08	KEB	RXE	Compare and Signal (Short)	B308	KEBR	RRE	Compare and Signal (Short)
ED18	KDB	RXE	Compare and Signal (Long)	B318	KDBR	RRE	Compare and Signal (Long)
				B348	KXBR	RRE	Compare and Signal (Extended)

Table 297. Binary floating-point Compare and Signal instructions

The CC settings for these instructions are the same as in Table 296 on page 662. As with the "normal" comparisons, masking off the invalid-operation interruption will set CC=3.

Figure 424 shows examples of binary floating-point compare and signal instructions.

```
LE    0,=EB'+0'         c(FPR0) = +0
KEB   0,=EB'-(Inf)'     CC=2 (0 > -infinity)
KEB   0,=EB'-0'         CC=0 (+0 = -0)
KEB   0,=EB'(QNaN)'     Invalid Operation; CC=3 if masked off
```

Figure 424. Examples of binary floating-point compare and signal instructions

Exercises

34.10.1.(2)+ Given operands A=+1, B=−0, C=+∞, D=−(Min), and E=−∞, show the Condition Code setting resulting from comparing each to the other four.

34.10.2.(2) Will the results you found in Exercise 34.10.1 be different if you use compare and signal instructions instead?

34.10.3.(4) A programmer claimed that two numeric short or long precision binary floating-point numbers X and Y satisfy the relation X < Y when compared as binary floating-point numbers *and* when they are compared as signed two's complement binary integers. Is this true or not?

34.11. Binary Floating-Point Rounding and Lengthening Instructions (*)

These two groups of instructions convert longer to shorter formats (with rounding) and shorter to longer formats.

34.11.1. Rounding Instructions (*)

The three Load Rounded instructions in Table 298 round a longer operand to a shorter form using the default Binary Rounding Mode in the FPCR. None of the instructions change the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B344	LEDBR	RRE	Load Rounded (Short←Long)	B346	LEXBR	RRE	Load Rounded (Short←Extended)
B345	LDXBR	RRE	Load Rounded (Long←Extended)				

Table 298. Binary floating-point Round instructions

Because each longer operand format has a wider exponent range than the range of the target operand, exponent overflow or underflow can be generated. (They are handled as described in Section 34.4.3 on page 651.) Figure 425 shows some examples:

```
         LD    0,BFPCon
         LD    2,BFPCon+8
*        c(FPR0,FPR2)=X'3FFF3C0CA428C59F B71A7BE16B6B6D43'
         LEXBR 4,0      Extended to short: c(FPR4)=X'3F9E0652'
         LDXBR 4,0      Extended to long:  c(FPR4)=X'3FF3C0CA428C59FB'
         LEDBR 4,0      Long to Short(??)  c(FPR4)=X'3FF9E065'?  (Wrong!)
         - - -
BFPCon   DC    LB'1.2345678901234567890123456789012З'
```

Figure 425. Examples of binary floating-point rounding instructions

The last example (using LEDBR) is invalid, because the second operand is the high-order half of an *extended-precision value*, while the LEDBR instruction expects a long precision second operand! Unlike hexadecimal floating-point operands, binary floating-point exponent field widths are different for each format, so you must be careful to ensure your instructions deal with the correct operand lengths.

The treatment of NaNs is interesting:

- The low-order bits of QNaNs are discarded. (The exponent field is adjusted to the proper length for the shorter format.)
- SNaNs cause an invalid-operation exception. If masked off, the result is the same as for QNaNs.

34.11.2. Lengthening Instructions (*)

The instructions in Table 299 on page 665 extend a shorter operand to a longer format, by adjusting the exponent and appending zeros to the low-order bits of the significand. The Condition Code is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED04	LDEB	RXE	Load Lengthened (Long←Short)	B304	LDEBR	RRE	Load Lengthened (Long←Short)
ED06	LXEB	RXE	Load Lengthened (Extended←Short)	B306	LXEBR	RRE	Load Lengthened (Extended←Short)
ED05	LXDB	RXE	Load Lengthened (Extended←Long)	B305	LXDBR	RRE	Load Lengthened (Extended←Long)

Table 299. Binary floating-point Lengthening instructions

These instructions are simpler than the Load Rounded instructions, because numeric values in a shorter format can always be represented correctly in a longer format. For example:

```
LE    0,=EB'0.1'   c(FPR0)=X'3DCCCCCD'
LDEBR 2,0          c(FPR2)=X'3FB99999A0000000'
LXDBR 4,2     c(FPR4,FPR6)=X'3FFB99999A000000 0000000000000000'
```

Figure 426. Examples of BFP load lengthened instructions

Figure 426 shows that lengthening an operand doesn't increase its accuracy, even though the longer value has greater precision.[223]

QNaNs are simply extended with zeros; SNaNs cause an invalid-operation exception, and place the corresponding QNaN in the first-operand register. Figure 427 shows some examples:

```
LE    0,=EB'(QNaN)'      c(FPR0)=X'7FE00000'  (Short QNaN)
LDEBR 2,0                c(FPR2)=X'7FFC000000000000 (Long QNaN)

LE    0,=EB'(SNaN)'      c(FPR0)=X'7FA00000'  (Short SNaN)
LDEBR 2,0                c(FPR2)=X'7FFC000000000000 (Long QNaN)
```

Figure 427. Examples of BFP load lengthened instructions with NaNs

Exercises

34.11.1.(1) Can the Load Rounded instructions be considered “shortening” or “truncating” instructions? Why or why not?

34.11.2.(2)+ Suppose you round a long binary floating-point operand X to short format, and then extend it to long format again. Estimate the difference between the original and final lengthened values.

34.11.3.(4)+ Suppose you execute these instructions for each of the short precision binary floating-point operands listed below:

```
LZDR  0                  Set FPR0 to zero
LE    0,data_item        Load a short data item
```

1. =EB'1.0'
2. =EB'(Min)'
3. =EB'(Dmin)'
4. =EB'(QNaN)'
5. =EB'(SNaN)'
6. =EB'(Inf)'

In each case, if you then treat the number in FPR0 as a *long* binary floating-point value, what will that long value be? What can you infer from the results?

[223] If you set a value like π to 9.876543210987654321..., the number may have great precision but no accuracy.

34.12. Converting Between BFP and Binary Integers (*)

System z instructions let you convert binary integers in the general registers to binary floating-point format, and vice versa. We'll start with the integer-to-float instructions.

34.12.1. Converting Binary Integers to Binary Floating-Point (*)

The instructions in Table 300 convert the binary integer in the second-operand general register to one of the three binary floating-point formats in the first-operand floating-point register, using the current rounding mode in the FPCR if needed. None of them affect the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B394	CEFBR	RRE	Convert from Fixed (Short←32)	B3A4	CEGBR	RRE	Convert from Fixed (Short←64)
B395	CDFBR	RRE	Convert from Fixed (Long←32)	B3A5	CDGBR	RRE	Convert from Fixed (Long←64)
B396	CXFBR	RRE	Convert from Fixed (Extended←32)	B3A6	CXGBR	RRE	Convert from Fixed (Extended←64)

Table 300. Binary integer to binary floating-point conversion instructions

Some examples are shown in Figure 428.

```
L     0,=F'2147483647'   c(GR0)= X'7FFFFFFF'
SRNM  1(0)               Set rounding mode to 'truncate'
CEFBR 4,0                c(FPR4)=X'4EFFFFFF', truncated
SRNM  0(0)               Set rounding mode to 'to nearest'
CEFBR 4,0                c(FPR0)=X'4F000000', rounded

LG    1,=FD'-99'         c(GG1)= X'FFFFFFFFFFFFFF9D'
CDGBR 6,1                c(FPR6)=X'C058C00000000000'
```

Figure 428. Examples of binary integer to binary floating-point instructions

Because the value in register GG1 is small enough, the result in FPR6 needed no rounding.

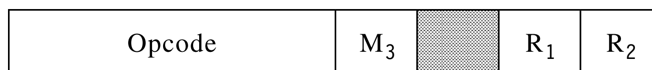
34.12.2. Converting Binary Floating-Point to Binary Integers (*)

These instructions convert binary floating-point operands in a floating-point register to a two's complement fixed-point binary value in a general register:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B398	CFEBR	RRF	Convert to Fixed (32←Short)	B3A8	CGEBR	RRF	Convert to Fixed (64←Short)
B399	CFDBR	RRF	Convert to Fixed (32←Long)	B3A9	CGDBR	RRF	Convert to Fixed (64←Long)
B39A	CFXBR	RRF	Convert to Fixed (32←Extended)	B3AA	CGXBR	RRF	Convert to Fixed (64←Extended)

Table 301. Binary floating-point to integer conversion instructions

The six instructions in Table 301 have an additional mask operand specifying a “local” rounding mode that takes effect only for that single instruction. Their format is illustrated in Table 302:



Opcode	M_3		R_1	R_2

Table 302. Format of BFP Convert To Fixed instructions

Even though the local rounding mode mask is called M_3, it appears in the machine instruction statement's operand field as the second operand:

```
Mnemonic  R1,M3,R2
```

As noted in Section 34.3 on page 646, the meanings of the local rounding mode field M_3 are:

M_3	Meaning
B'0000'	Round according to the current "global" rounding mode in the FPCR
B'0001'	Biased round to nearest (ties round away from zero; the same as adding 1 to the first lost bit)
B'0100'	Unbiased round to nearest
B'0101'	Round toward zero (truncate)
B'0110'	Round toward $+\infty$
B'0111'	Round toward $-\infty$

Table 303. Rounding modifier for BFP convert to fixed instructions

Local rounding mode 1 is the traditional "round up" action.

The effect of the rounding modifiers on fraction-to-integer conversion is illustrated in Figure 429.

```
Up:            + 3.5  ──►  + 4,    − 3.5  ──►  − 3
Down:          + 3.5  ──►  + 3,    − 3.5  ──►  − 4
To Zero:       + 3.5  ──►  + 3,    − 3.5  ──►  − 3
To Nearest:    ± 3.4  ──►  ± 3,    ± 2.6  ──►  ± 3
To Nearest:    ± 3.5  ──►  ± 4,    ± 2.5  ──►  ± 2 (ties to even)
Biased Round:  ± 3.5  ──►  ± 4,    ± 2.5  ──►  ± 3 (away from zero)
```

Figure 429. Examples of converting binary floating-point fractions to integers with rounding

To obtain these results we can use the instructions in Figure 430:

```
LE    8,=EB'3.5'         c(FPR8)=X'40600000'
CFEBR 11,B'0110',8       c(GR11)=X'00000004' (Round up)
CFEBR 12,B'0111',8       c(GR12)=X'00000003' (Round down)
CFEBR 13,B'0101',8       c(GR13)=X'00000003' (Round toward 0)
CFEBR 14,B'0100',8       c(GR14)=X'00000004' (To nearest even)
CFEBR 15,B'0001',8       c(GR15)=X'00000004' (Biased round)
LE    6,=EB'2.5'         c(FPR6)=X'40200000'
CFEBR 10,B'0100',6       c(GR10)=X'00000002' (To nearest even)
```

Figure 430. Examples of Convert to Fixed instructions

The Condition Code is set as shown in Table 304:

CC	Meaning
0	Source was zero
1	Source was less than zero
2	Source was greater than zero
3	Special case

Table 304. CC settings after convert to binary instructions

These are the special cases resulting in CC=3:

- The size of the binary floating-point value lies outside the representable range of a 32- or 64-bit integer; an invalid operation exception is signaled.
  - If the "Inexact result" mask bit is zero, CC3 is set, and the maximum correctly signed integer is placed in the target R_1 general register.

- If the mask bit is one, a program interruption occurs.

- If the source operand is a NaN and the invalid-operation mask bit is 1, a program interruption occurs; otherwise, the target R_1 general register is set to the maximum negative number.

An inexact exception may also occur, but these interruptions are typically masked off.

These instructions make it very easy to choose the rounding you want to use for converting binary floating-point values to binary integers.

Exercises

34.12.1.(2) An array of fullword integers is stored starting at **IntData** and the number of elements in the list is stored as a fullword integer at **NItems**. Write an instruction sequence to compute and store at **IntAvg** the binary floating-point average of the list of integers. Take into account the possibility that the integer sum may overflow a general register.

34.12.2.(1)+ Can the instructions for converting a binary integer to binary floating-point generate an exponent underflow or overflow?

34.12.3.(2)+ What integer values satisfying

$2^{30} \leq \text{value} < 2^{31}$

can be converted without loss of precision to short binary floating-point form?

34.13. Binary Floating-Point Integers and Remainders (*)

System z provides two groups of instructions for deriving integer values of binary floating-point numbers, and for calculating remainders.

As with other binary floating-point instructions, a QNaN operand leads to a QNaN result, and a SNaN operand causes an invalid operation exception. If masked off, the result is the corresponding QNaN.

34.13.1. Load FP Integer Instructions

The three Load FP Integer instructions in Table 305 round a second-operand binary floating-point operand to a first-operand integer value in the same format.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B357	FIEBR	RRF	Load FP Integer (Short)	B35F	FIDBR	RRF	Load FP Integer (Long)
B347	FIXBR	RRF	Load FP Integer (Extended)				

Table 305. Load floating-point integer instructions

The three instructions have the format shown in Table 302 on page 666. The format of the machine instruction statement is

```
mnemonic  R1,M3,R2
```

where M_3 is a rounding modifier with values shown in Table 306 on page 669:

M_3	Meaning
B'0000'	Round according to the current BFP rounding mode
B'0001'	Round to nearest with ties away from 0
B'0100'	Round to nearest with ties to even
B'0101'	Round toward 0
B'0110'	Round toward $+\infty$
B'0111'	Round toward $-\infty$

Table 306. Rounding mode modifiers for BFP load integer instructions

If we execute the instructions in Figure 431, the results in FPR2 are as shown:

```
LE    0,=EB'5.6789012'   c(FPR0)=X'40B5B98F'
FIEBR 2,B'0100',0        c(FPR2)=X'40C00000' (Round to nearest)
FIEBR 2,B'0001',0        c(FPR2)=X'40C00000' (Round away from 0)
FIEBR 2,B'0101',0        c(FPR2)=X'40A00000' (Round toward 0)
FIEBR 2,B'0111',0        c(FPR2)=X'40A00000' (Round down)
```

Figure 431. Examples of load FP integer instructions

The first and second results have value +6, and the third and fourth results have value +5.

Because the initial operand in FPR0 is not already an integer, there will be an exception condition for an inexact result; if not masked off, and an interruption will occur.

Remember

These instructions produce a *binary floating-point* integer-valued result in a floating-point register, not a binary integer in a general register.

34.13.2. Divide to Integer Instructions (*)

Table 307 lists the two instructions[224] used to calculate binary floating-point remainders:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B353	DIEBR	RRF	Divide to Integer (Short)	B35B	DIDBR	RRF	Divide to Integer (Long)

Table 307. Binary floating-point Divide to Integer instructions

The format of these two instructions is shown in Table 308:

Opcode	R_3	M_4	R_1	R_2

Table 308. Format of BFP Divide to Integer instructions

The format of the machine instruction statement is

mnemonic R_1,R_3,R_2,M_4

where all the register operands must be different, and M_4 is a rounding modifier with the same values shown in Table 306 above. Note that only the *final* quotient is rounded.

The first operand is divided by the second operand, and the integer quotient replaces the third operand; the remainder replaces the dividend in the first operand.

If you divide x by y, a floating-point remainder is defined by the relation

$$r = x-(y\times n)$$

224 These instructions are complex; before using them you should study their description in the *z/Architecture Principles of Operation*.

where n is the integer nearest the *exact* value of x/y.

For example, suppose we want the remainder of 6.5/2.0. The quotient is 3 and the remainder is 0.5. However, if we want the remainder of 7.5/2.0, the “true” quotient n is again 3 and the “true” remainder is 1.5. Depending on the rounding mode chosen for the final quotient, the remainder could be -0.5! This shows why:

```
6.5/2.0 with rounding toward zero:     qexact=3.25, so q=3.0, r=+0.5
7.5/2.0 with rounding toward zero:     qexact=3.75, so q=3.0, r=+1.5

6.5/2.0 with rounding toward nearest:  qexact=3.25, so q=3.0, r=+0.5
7.5/2.0 with rounding toward nearest:  qexact=3.75, so q=4.0, r=-1.5
```

The instructions in Figure 432 show this behavior (where the rounding mode is “to nearest”):

```
         LE    0,=EB'6.5'         c(FPR0)=X'40D00000'
         LE    2,=EB'2.0'         c(FPR0)=X'40000000'
         DIEBR 0,4,2,B'0100'      c(FPR0)=X'3F000000' (Remainder=+0.5)
*                                 c(FPR4)=X'40400000' (Quotient =+3.0)

         LE    0,=EB'7.5'         c(FPR0)=X'40F00000'
         DIEBR 0,4,2,B'0100'      c(FPR0)=X'BF000000' (Remainder=-0.5)
*                                 c(FPR4)=X'40800000' (Quotient =+4.0)
```

Figure 432. Examples of divide to integer instructions

Sometimes the calculation of a remainder is lengthy, so the CPU sets the CC to indicate that the operation is incomplete, as we saw for instructions like MVCLE and CLCLE in Section 25. The CC settings are shown in Table 309:

CC	Meaning
0	Remainder complete, normal quotient
1	Remainder complete, quotient overflow or NaN
2	Remainder incomplete, normal quotient
3	Remainder incomplete, quotient overflow or NaN

Table 309. CC settings after divide to integer instructions

These CC settings are somewhat unusual. Rather than indicating an exception condition for quotient overflow, CC values 1 and 3 indicate that the exponent of the quotient has been scaled. CC values 2 or 3 mean you should repeat the instruction until the remainder is complete.

If a quotient overflow occurs or if an operand is a SNaN, an invalid operation exception is indicated. If masked off, the result is a QNaN.

To illustrate, suppose we execute the instructions in Figure 433:

```
         LE    8,=EB'1E28'        Dividend is large, X'6E013F39'
         LE    3,=EB'.73E-9'      Divisor is small, X'3048A92F'
Div      DIEBR 8,0,3,B'0100'      Partial quotient in FPR0,
*                                 partial remainder in FPR8
         BC    3,Div              Iterate if not complete
```

Figure 433. Example of iterative divide to integer

A final result requires five iterations. The CC setting, the values in FPR8 and FPR0, and the partial remainders (PR) and partial quotients (PQ) are developed in these steps:

```
                CC    FPR8         FPR0
Execution 1    2  X'60985BC8'  X'7D24E429'  PR=8.78E19     PQ=1.37E37
Execution 2    2  X'54A2FBB9'  X'6FC26069'  PR=5.60E12     PQ=1.20E29
Execution 3    2  X'48A5FFF8'  X'63CFEE78'  PR=3.40E5      PQ=7.67E21
Execution 4    2  X'399F6040'  X'57D3C7B1'  PR=3.04E-4     PQ=4.66E14
Execution 5    0  X'AFB07DDA'  X'48CB5460'  PR=-3.21E-10   PQ=4.16E5
```

Figure 434. Iterative execution of a divide to integer instruction

Calculating the quotient manually gives a value 1.3698E37, as we see in the partial quotient of the first iteration.

Exercises

34.13.1.(2)+ Show the result in FPR2 of executing

```
LE    0,=EB'3.5'
FIEBR 2,mask,0
```

for mask values 1, 4, 5, 6, and 7.

34.13.2.(3)+ What result will appear in FPR2 after executing

```
LD    0,=DB'9.5'
FIEBR 2,4,0
```

Be careful!

34.13.3.(4) Show that the rule requiring the generation of an even quotient in the case when $|n\text{-}x/y|=1/2$ and the rounding mode is “to nearest even” leads to a remainder satisfying $|r| \leq y/2$.

34.14. Binary Floating-Point Square Root Instructions (*)

Four of the square root instructions in Table 310 have both RX- and RR-type forms; SQXBR has only RR format.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED14	SQEB	RXE	Square Root (Short)	B314	SQEBR	RRE	Square Root (Short)
ED15	SQDB	RXE	Square Root (Long)	B315	SQDBR	RRE	Square Root (Long)
				B316	SQXBR	RRE	Square Root (Extended)

Table 310. Binary floating-point Square Root instructions

The square root of the binary floating-point second operand is evaluated, rounded according to the current BFP rounding mode, and placed in the R_1 floating-point register. Negative nonzero values and SNaNs cause an invalid-operation exception; if masked off, the result is a QNaN. The square root of $+\infty$ is $+\infty$.[225]

Figure 435 on page 672 shows some examples of the binary floating-point square root instructions.

[225] Seems reasonable.

```
         SQEB  0,=EB'10'          c(FPR0)=X'404A62C2'
         SQDB  2,=DB'9999'        c(FPR2)=X'4058FFAE13F4A7D3'
         LD    4,=LB'0.1'         c(FPR4)=X'3FFB999999999999'
         LD    6,=LB'0.1'+8       c(FPR6)=X'999999999999999A'
         SQXBR 4,4                c(FPR4)=X'3FFD43D136248490',
*                                 c(FPR6)=X'EDB36E896CF3D7B0'
         SQEB  0,=EB'(QNaN)'      c(FPR0)=X'7FE00000'  (QNaN)
         SQEB  2,=EB'(Inf)'       c(FPR2)=X'7F800000'  (+infinity)
```

Figure 435. Examples of binary floating-point square root instructions

Exercises

34.14.1.(1)+ Estimate the value of the square root of BFP (Max), (Min), and (DMin) values.

34.14.2.(2)+ What result will appear in FPR0 if you execute this instruction?

```
SQEB  0,=DB'16'
```

Be careful!

34.15. Binary Floating-Point Multiply and Add/Subtract (*)

Table 311 lists the binary floating-point “multiply and add” and “multiply and subtract” instructions. The Condition Code is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED0E	MAEB	RXF	Multiply and Add (Short)	B30E	MAEBR	RRF	Multiply and Add (Short)
ED1E	MADB	RXF	Multiply and Add (Long)	B31E	MADBR	RRF	Multiply and Add (Long)
ED0F	MSEB	RXF	Multiply and Subtract (Short)	B30F	MSEBR	RRF	Multiply and Subtract (Short)
ED1F	MSDB	RXF	Multiply and Subtract (Long)	B31F	MSDBR	RRF	Multiply and Subtract (Long)

Table 311. Binary floating-point Multiply and Add/Subtract instructions

The Assembler instruction statement format for these instructions is

R_1,R_3,R_2 For RRF-format
$R_1,R_3,D_2(X_2,B_2)$ For RXF-format

These instructions provide a more precise result for a very common mathematical operation:

operand1 = (operand3 × operand2) ± operand1

The product is formed internally to double length without rounding, and then operand1 is added or subtracted. This result is then rounded to the target operand length according to the current BFP rounding mode. This means that

- the product is more precise than would be delivered by a single multiplication operation, and
- the first operand is added to an unrounded product, with possibly more bits participating.

To illustrate the difference between using multiply and add compared to using a multiply and then an add instruction, consider the instructions shown in Figure 436 on page 673:

```
         LE    2,Operand2         c(FPR2)=Operand2 (and Operand3)
         LE    6,Operand1         c(FPR6)=Operand1
         MAEBR 6,2,2              c(FPR0)=Operand1+(Operand2**2)
*
         LE    2,Operand2         c(FPR2)=Operand2 (and Operand3)
         MEEBR 2,2                c(FPR2)=Operand2**2
         AEB   2,Operand1         c(FPR2)=Operand1+(Operand2**2)
```

Figure 436. Example of binary floating-point multiply and add instructions

In many cases, the results will differ by very little, perhaps by a single bit. But if these operations are part of a long loop involving sums of many products, the accumulated differences can be significant.

If the operands are close to overflow or underflow threshholds, the results can be quite different. For example:

```
         LE    0,=EB'1.85E19'     c(FPR0) = X'5F805E9A'
         MEEBR 0,0                c(FPR0) = X'1F80BD7A' (overflow)
         AEB   0,=EB'-(Max)'      c(FPR0) = X'FF7FFFFF' -(Max)
*
         LE    0,=EB'1.85E19'     c(FPR0) = X'5F805E9A'
         LE    2,=EB'-(Max)'      c(FPR2) = X'FF7FFFFF'
         MAEBR 2,0,0              c(FPR2) = X'7BBD7A6B'
```

The first three instructions multiply an operand slightly larger than the square root of (Max), generating an overflowed result with characteristic wrap. Subtracting (Max) overwhelms the wrapped result, leaving −(Max) as the result. The second three instructions add −(Max) to the intermediate result, generating a finite (and correct) result.

If any operand is a QNaN, the result is a QNaN taken from the original operands in order of precedence operand3, operand2, operand1. Any SNaN causes an invalid operation exception; if it is masked off, a default QNaN is delivered as the result.

Exercises

34.15.1.(2)+ Rewrite the example in Figure 387 on page 608 to use binary floating-point data and multiply-add instructions.

34.15.2.(2) Rewrite the example in Figure 388 on page 608 to use binary floating-point data and multiply-add instructions.

34.16. Summary

The binary floating-point instructions for several operation types are summarized in Table 312. All of these operations use operands of uniform lengths.

Table 312 (Page 1 of 2). Summary of binary floating-point instructions with uniform operand lengths

Function	Operand Length		
	4 bytes	**8 bytes**	**16 bytes**
Add/Subtract (register)	AEBR SEBR	ADBR SDBR	AXBR SXBR
Add/Subtract (storage)	AEB SEB	ADB SDB	
Compare (register)	CEBR	CDBR	CXBR
Compare (storage)	CEB	CDB	

Table 312 (Page 2 of 2). Summary of binary floating-point instructions with uniform operand lengths

Function	Operand Length		
	4 bytes	**8 bytes**	**16 bytes**
Divide (register)	DEBR	DDBR	DXBR
Divide (storage)	DEB	DDB	
Divide to Integer	DIEBR	DIDBR	
Compare and Signal (register)	KEBR	KDBR	KXBR
Compare and Signal (storage)	KEB	KDB	
Load Positive (register)	LPEBR	LPDBR	LPXBR
Load Negative (register)	LNEBR	LNDBR	LNXBR
Load Complement (register)	LCEBR	LCDBR	LCXBR
Load and Test (register)	LTEBR	LTDBR	LTXBR
Load FP Integer	FIEBR	FIDBR	FIXBR
Multiply and Add/Subtract (register)	MAEBR MSEBR	MADBR MSDBR	
Multiply and Add/Subtract (storage)	MAEB MSEB	MADB MSDB	
Square Root (register)	SQEBR	SQDBR	SQXBR
Square Root (storage)	SQEB	SQDB	
Test Data Class (register)	TCEB	TCDB	TCXB

Tables 313 through 317 summarize instructions whose operands may have mixed lengths.

The binary floating-point multiplication instructions are summarized in Table 313.

Func-tion	Source Length	4 bytes		8 bytes		16 bytes
	Product Length	**4 bytes**	**8 bytes**	**8 bytes**	**16 bytes**	**16 bytes**
Multiply (registers)		MEEBR	MDEBR	MDBR	MXDBR	MXBR
Multiply (storage)		MEEB	MDEB	MDB	MXDB	

Table 313. Binary floating-point Multiply instructions

The binary floating-point rounding instructions are summarized in Table 314.

Function	Source Length	8 bytes	16 bytes	16 bytes
	Result Length	**4 bytes**	**4 bytes**	**8 bytes**
Round		LEDBR	LEXBR	LDXBR

Table 314. Binary floating-point Round instructions

The binary floating-point operand-lengthening instructions are summarized in Table 315.

Function	Source Length	4 bytes		8 bytes
	Result Length	**8 bytes**	**16 bytes**	**16 bytes**
Lengthen (register)		LDEBR	LXEBR	LXDBR
Lengthen (storage)		LDEB	LXEB	LXDB

Table 315. Binary floating-point Lengthening instructions

The instructions for converting binary floating-point operands to binary integers are summarized in Table 316 on page 675.

Function	Source Length	4 bytes		8 bytes		16 bytes	
	Target Length	32 bits	64 bits	32 bits	64 bits	32 bits	64 bits
Convert float to binary		CFEBR	CGEBR	CFDBR	CGDBR	CFXBR	CGXBR

Table 316. Convert binary floating-point to binary integer instructions

The instructions for converting binary integer operands to binary floating-point are summarized in Table 317.

Function	Source Length	32 bits			64 bits		
	Target Length	4 bytes	8 bytes	16 bytes	4 bytes	8 bytes	16 bytes
Convert binary to float		CEFBR	CDFBR	CXFBR	CEGBR	CDGBR	CXGBR

Table 317. Convert binary integer to binary floating-point instructions

Table 318 summarizes the binary floating-point exception conditions that might be caused by the instructions described in this section. F denotes a finite value, and IMax is the largest representable binary integer in the target format.

Operation	Invalid Operation	Divide by Zero	Overflow	Underflow	Inexact
Add, Subtract	SNaN, $\infty - \infty$	—	Yes	Yes	Yes
Compare, Compare and Signal	SNaN	—	—	—	—
Convert to Fixed	NaN, $\|F\|>\|IMax\|$	—	—	—	Yes
Divide	SNaN, $0 \div 0$, $\infty \div \infty$	$F \div 0$	Yes	Yes	Yes
Divide to Integer	SNaN, Any$\div \infty$	—	—	Yes	Yes
Load and Test	SNaN	—	—	—	—
Load FP Integer	SNaN	—	—	—	Yes
Load Lengthened	SNaN	—	—	—	—
Load Rounded	SNaN	—	Yes	Yes	Yes
Multiply	SNaN, $\infty \times 0$	—	Yes	Yes	Yes
Multiply & Add, Multiply & Subtract	SNaN, $\infty - \infty$, $\infty \times 0$	—	Yes	Yes	Yes
Square Root	SNaN, $-\infty$, F<0	—	—	—	Yes
Note: F is any finite value; IMax is the largest available signed integer in the target format.					

Table 318. Summary of binary floating-point operations and exceptions

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
ADB	ED1A
ADBR	B31A
AEB	ED0A
AEBR	B30A
AXBR	B34A
CDB	ED19
CDBR	B319
CDFBR	B395
CDGBR	B3A5
CEB	ED09
CEBR	B309
CEFBR	B394
CEGBR	B3A4
CFDBR	B399
CFEBR	B398
CFXBR	B39A
CGDBR	B3A9
CGEBR	B3A8
CGXBR	B3AA
CXBR	B349
CXFBR	B396
CXGBR	B3A6
DDB	ED1D
DDBR	B31D
DEB	ED0D
DEBR	B30D
DIDBR	B35B
DIEBR	B353
DXBR	B34D
EFPC	B38C
FIDBR	B35F
FIEBR	B357

Mnemonic	Opcode
FIXBR	B347
KDB	ED18
KDBR	B318
KEB	ED08
KEBR	B308
KXBR	B348
LCDBR	B313
LCEBR	B303
LCXBR	B343
LDEB	ED04
LDEBR	B304
LDXBR	B345
LEDBR	B344
LEXBR	B346
LFAS	B2BD
LFPC	B29D
LNDBR	B311
LNEBR	B301
LNXBR	B341
LPDBR	B310
LPEBR	B300
LPXBR	B340
LTDBR	B312
LTEBR	B302
LTXBR	B342
LXDB	ED05
LXEB	ED06
LXEBR	B306
LXDBR	B305
MADB	ED1E
MADBR	B31E
MAEB	ED0E

Mnemonic	Opcode
MAEBR	B30E
MDB	ED1C
MDBR	B31C
MDEB	ED0C
MDEBR	B30C
MEEB	ED17
MEEBR	B317
MSDB	ED1F
MSDBR	B31F
MSEB	ED0F
MSEBR	B30F
MXBR	B34C
MXDB	ED07
MXDBR	B307
SDB	ED1B
SDBR	B31B
SEB	ED0B
SEBR	B30B
SFASR	B385
SFPC	B384
SQDB	ED15
SQDBR	B315
SQEB	ED14
SQEBR	B314
SQXBR	B316
SRNM	B299
STFPC	B29C
SXBR	B34B
TCDB	ED11
TCEB	ED10
TCXB	ED12

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
B299	SRNM
B29C	STFPC
B29D	LFPC
B2BD	LFAS
B300	LPEBR
B301	LNEBR
B302	LTEBR
B303	LCEBR
B304	LDEBR
B305	LXDBR
B306	LXEBR
B307	MXDBR
B308	KEBR
B309	CEBR
B30A	AEBR
B30B	SEBR
B30C	MDEBR
B30D	DEBR
B30E	MAEBR
B30F	MSEBR
B310	LPDBR
B311	LNDBR
B312	LTDBR
B313	LCDBR
B314	SQEBR
B315	SQDBR
B316	SQXBR
B317	MEEBR
B318	KDBR
B319	CDBR
B31A	ADBR
B31B	SDBR

Opcode	Mnemonic
B31C	MDBR
B31D	DDBR
B31E	MADBR
B31F	MSDBR
B340	LPXBR
B341	LNXBR
B342	LTXBR
B343	LCXBR
B344	LEDBR
B345	LDXBR
B346	LEXBR
B347	FIXBR
B348	KXBR
B349	CXBR
B34A	AXBR
B34B	SXBR
B34C	MXBR
B34D	DXBR
B353	DIEBR
B357	FIEBR
B35B	DIDBR
B35F	FIDBR
B384	SFPC
B385	SFASR
B38C	EFPC
B394	CEFBR
B395	CDFBR
B396	CXFBR
B398	CFEBR
B399	CFDBR
B39A	CFXBR
B3A4	CEGBR

Opcode	Mnemonic
B3A5	CDGBR
B3A6	CXGBR
B3A8	CGEBR
B3A9	CGDBR
B3AA	CGXBR
ED04	LDEB
ED05	LXDB
ED06	LXEB
ED07	MXDB
ED08	KEB
ED09	CEB
ED0A	AEB
ED0B	SEB
ED0C	MDEB
ED0D	DEB
ED0E	MAEB
ED0F	MSEB
ED10	TCEB
ED11	TCDB
ED12	TCXB
ED14	SQEB
ED15	SQDB
ED17	MEEB
ED18	KDB
ED19	CDB
ED1A	ADB
ED1B	SDB
ED1C	MDB
ED1D	DDB
ED1E	MADB
ED1F	MSDB

Terms and Definitions

Data Exception Code
A field in the FPCR indicating which of various floating-point and packed decimal exceptions have occurred.

denormalized number
A nonzero binary floating-point value with characteristic zero and a nonzero fraction.

DXC
Data Exception Code, a field in the FPCR.

exception condition
One of five conditions defined by the IEEE Floating-Point Standard: invalid operation, division by zero, exponent overflow, exponent underflow, and inexact result.

Floating-Point Control Register (FPCR)
A special register containing IEEE masks, status indicators, Data Exception Code, and rounding modes.

gradual underflow
A technique allowing numbers to become denormalized when they are finite and smaller than the smallest normalized magnitude.

mask
A bit in the FPCR controlling the actions to be taken when an exception condition occurs.

NaN
A “Not-a-Number” having no numeric or mathematical meaning.

QNaN
A Quiet Not-a-Number that does not cause an exception condition in any arithmetic operation.

Rounding Mode
A field in the FPCR indicating the rounding action to be taken after a binary floating-point operation.

rounding modifier
A field in an instruction specifying the type of rounding to be performed on the result of the operation.

SNaN
A Signaling Not-a-Number that causes an exception condition in an arithmetic operation.

special value
A zero, a denormalized number, an infinity, a QNaN, or an SNaN.

status flag
A bit in the FPCR indicating that an exception condition has occurred.

Programming Problems

Problem 34.1.(2) Write a program using binary floating-point operands to generate each of the five exception conditions. First, set the five mask bits to zero, and see what default results are delivered for each exception type; then, generate the same exceptions with each mask bit set to one. Determine the ways your operating system reports the interruptions.

Problem 34.2.(4) Write a program that will read records containing eight hex digits representing short binary floating-point numbers. Then, print the approximate (but reasonably accurate) decimal value of the binary floating-point number to five significant digits, in the form `s.dddddEtnn`, where s is the sign of the number, ddddd are the significant decimal digits, t is the sign of the exponent, and nn is the decimal exponent. If the number is a NaN or infinity, print an appropriate indication. (See Problem 33.3.)

Problem 34.3.(2) Write a program that calculates a table of the square roots of short precision binary floating-point integer values from 1 to 20. For each square root value, also calculate its square and compare the result to the original integer value. Then, display the original value, its square root, and the difference between the integer value and the squared square root. (If you can display the results as decimal values, perhaps using your solution to Problem 34.2, so much the better!)

Problem 34.4.(3) A programmer claimed that evaluating (1.0/N)*N in binary floating-point arithmetic (with (default) rounding to nearest even) for values of N from 1 to 100 does not always produce 1.0. Write a program that will test this claim in short, long, and extended binary floating-point arithmetic.

Problem 34.5.(3) Using the iterative technique described in Section 31.3.1 on page 554, write a program to evaluate the square root of 2 to 10 significant digits using long binary floating-point arithmetic, and without using any square-root instructions. Format and print the result as a fixed-point value.

35. Decimal Floating-Point Data and Operations

```
 3333333333    55555555555
333333333333  555555555555
33        33  55
          33  55
          33  555555555
        3333   5555555555
        3333          555
          33           55
          33           55
33        33          555
333333333333  55555555555
 3333333333   555555555
```

Having seen in Sections 33 and 34 that System z supports both hexadecimal and binary floating-point, you might ask why yet another is needed.

- We are ten-fingered creatures: (almost) all humans count in decimal, and non-decimal arithmetic is highly unintuitive for most people.[226]
- When calculations using hexadecimal or binary floating-point are compared with "hand" calculations in decimal, the results may be different. Decimal notation and arithmetic is pervasive in business use, and is often a legal requirement.
- If your processor is capable of scaled fixed-point decimal arithmetic (like packed decimal), such computations are difficult because the position of the decimal point must be remembered.
  - Even when you *can* do this the resulting code may be complex, difficult to understand and maintain, and slow.
- One of the most annoying aspects of non-decimal arithmetic is that representations of decimal fractions (e.g., 1/10) are imprecise and must be rounded, which can cause accumulated errors.
  - Laws in many countries require exact decimal rounding for financial calculations.
- Conversions between decimal and hexadecimal or binary are complicated, and can be quite difficult to do correctly. Imprecise conversions can lead in either binary or packed decimal to many other problems.
- Floating-point remainders are often not what you would expect in bases other than ten.
- The decimal "precision" of a binary or hexadecimal floating-point number is often not what it seems. The question "How many significant digits?" almost always implies *decimal* digits.
- The results of floating-point comparisons can be surprising: two values that you think should be algebraically equal may not be.
- Decimal floating-point is now an international standard (IEEE Std 754™-2008).

[226] Old joke: "There are 10 types of people who understand binary: those who do, and those who don't."

A desired solution should have these properties:

- Intuitive, familiar decimal arithmetic
- Exact representation of (most) decimal numbers
  - Problems with repeating non-decimal fractions like 1/3 are well known and understood.
- Automatic tracking of the decimal point's placement
- No conversions between decimal and hex or binary are needed, so there are fewer misconceptions about the decimal precision of a number.
- Rounding uses decimal rules, not binary or hexadecimal.
  - More rounding modes are supported.
- Low-order digits are preserved whenever possible.
- Integer, fixed point, and floating-point values are supported without extra effort.
  - Businesses need all three, for things like counts, currency, and interest and tax rates.

As we will see, the System z decimal floating-point data types and instructions have these properties.

Exercises

35.0.1.(0) Many cartoon characters are drawn with only three fingers and a thumb on each hand. Use that property to devise examples for teaching children octal (base-8) arithmetic.

35.0.2.(1) What other number bases are in widespread use today?

35.1. Representations

The significand in the binary and hexadecimal floating-point representations is a string of hexadecimal or binary digits, packaged with a sign and an exponent:

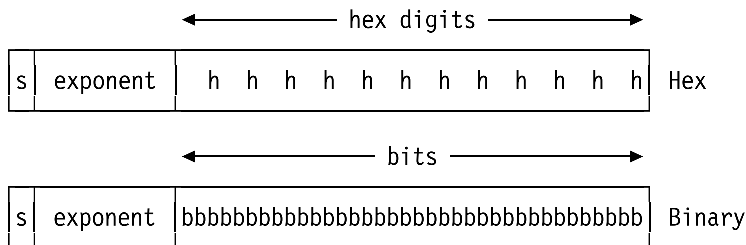
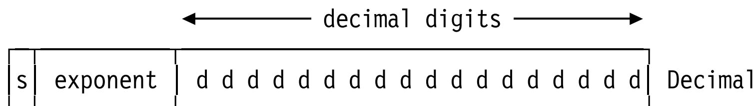


Figure 437. Hexadecimal and binary floating-point representations

You might reasonably expect something similar for decimal:



Three possible representations could be considered:

1. Like hexadecimal floating-point, the significand could contain BCD (not EBCDIC!) digits.[227]

   This has some advantages and disadvantages:

   - No conversion of external decimal data to and from binary or hexadecimal is needed.

[227] One of the earliest time-sharing systems on System/360 was known as RUSH; its decimal floating-point representation used BCD rather than hexadecimal digits, and its implementation used part of the packed decimal microcode.

- Rounding can use similar techniques as for the packed decimal instruction SRP (discussed in Section 29.7).
- Because the operations would be done in the floating-point registers rather than in storage, they could be faster than for packed decimal.
- The representation is somewhat wasteful, using only 62.5 percent of the available “bit space” (digits A-F are not present). Encoding decimal values with BCD digits is inefficient because each 3 BCD digits require 12 bits, but values from 000 to 999 can be stored in 10 bits.

2. The significand could be a binary integer, as sketched in this figure:

This has some advantages and disadvantages:

- The representation is compact; no significand bit combinations are disallowed.
- Arithmetic should be about as fast as for the binary floating-point instructions.
- Data would require conversion from BCD formats.
- A serious disadvantage is that decimal rounding, shifting, and scaling are complicated and possibly expensive.

This format is actually defined by the IEEE standard, but is not supported by System z. (If you're interested, see Section 35.14 on page 730.)

3. The significand could contain compressed BCD digits.

This has some advantages and disadvantages:

- The representation can be made very efficient, having the compactness of binary and the ease of scaling, shifting, and rounding of decimal.
- The encoding is complex and quite unintuitive.
- Decimal operations are natural.

The encoding chosen for System z to represent decimal floating-point data is the last one.

35.1.1. Conceptual View of the Decimal Floating-Point Representation

It's easiest to think of decimal floating-point data having the representation shown in Figure 438. As with hexadecimal and binary floating-point representations, the exponent field contains a characteristic (the exponent plus a bias to ensure that it's nonnegative).

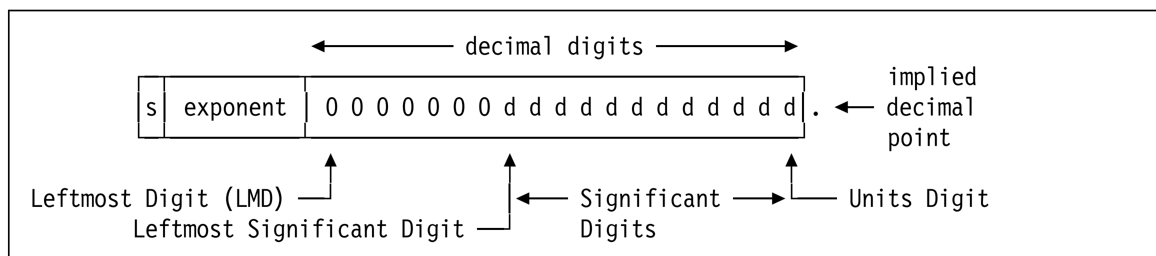


Figure 438. Conceptual decimal floating-point representation

An important property of decimal floating-point data is that it is *not* always normalized, either as a fraction or as an integer. This means that it is easy to generate equal values that don't have the same representation. For example, 1×10^3 and 100×10^1 have the same value but different representations.

The set of redundant representations with the same value is called a “cohort”:[228] each such number is a “cohort member”.

To illustrate a decimal floating-point cohort, consider the notation we used in Section 32.9, where a 4-digit decimal floating-point number is represented as [exponent‖significand]. The decimal number 100 then has four possible representations: [−1‖1000], [0‖0100], [+1‖0010], and [+2‖0001]; this cohort has 4 members.

None of the values is necessarily normalized, either as a fraction (as in the FPF(10,4) representation) nor as an integer (as in the FPI(10,4) representation), even though the first and last cohort members appear to be left- and right-normalized.

A new term is introduced in place of the “ulp” (unit in the last place) we saw in discussing hexadecimal and binary floating-point: the units-digit value, $1 \times 10^{\text{exponent}}$, is called the *quantum*.

Remember!

Finite numbers with the same exponent have the same quantum.

Each of the four representations above has a different quantum, respectively 1×10^{-1}, 1×10^{0}, $1\times10^{+1}$, and $1\times10^{+2}$. This shows how the quantum of a value depends on which member of a cohort is chosen.

Comment

The *z/Architecture Principles of Operation* uses “quantum” in describing decimal floating-point data and operations. It's much simpler to think of a decimal floating-point value as an integer with an exponent (as in Figure 438 on page 682).

For example, the three numbers in Figure 439 represent the same value, 12345×10^{7}, so they are members of the same cohort. But they have different exponents (and therefore, different quanta).

```
|+|      7 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5|

|+|      6 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 0|

|+|      3 | 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 0 0 0 0|
```

Figure 439. Three decimal floating-point representations of the same value

You may remember from Section 27.4 on page 467 that a packed decimal number can have multiple representations with the same value, and that different values can have the same representation. Different decimal floating-point values always have different representations.

Table 319 on page 684 summarizes the properties of the System z decimal floating-point representations, where approximations are given for Max, Min, and DMin values. Because decimal floating-point numbers aren't usually normalized, the column headed “Max Norm.” refers to values with nonzero leftmost digit, and the two “Min” columns refer to values with only 1 in the units digit.

[228] The term “cohort” actually refers to a division of 300 to 600 troops of a Roman army legion. Its informal meaning of a group of associates (not an associate of a “hort”) is intended here.

Byte length	Char. (bits)	Min exp.	Max exp.	Char. Bias	Pre-cision	Max Norm. (Max)	Min Norm. (Min)	Min Denorm. (DMin)
4	8	−95	+96	+101	7	$1.0\times10^{+97}$	1.0×10^{-95}	1.0×10^{-101}
8	10	−383	+384	+398	16	$1.0\times10^{+385}$	1.0×10^{-383}	1.0×10^{-398}
16	14	−6143	+6144	+6176	34	$1.0\times10^{+6145}$	1.0×10^{-6143}	1.0×10^{-6176}

Table 319. Decimal floating-point data representations

The "1.0×10^N" values in Table 319 are actually $0.9999...\times10^N$, which were rounded for simplicity.

Decimal floating-point has many advantages over packed decimal:

- Its behavior is governed by industry standards.
- It keeps track of the decimal point automatically.
- Fewer programming languages support packed decimal.
- Its performance is faster because arithmetic is done in registers, rather than in memory.
- Rounding support is far more varied than the simple "round-by-adding" of the SRP instruction.

Exercises

35.1.1.(2) Short hexadecimal and binary floating-point data can correctly represent only 6 decimal digits. Assuming a 7-bit exponent, show how you would create a 24-bit decimal floating-point data item with BCD digits used for the significand. What should be the minimum and maximum values of the exponent, EMax and EMin, and what should be the characteristic bias?

35.1.2.(2)+ In Figure 439, what is the quantum of each value?

35.1.3.(1)+ Suppose two decimal floating-point numbers have representations of 7 and 16 digits respectively. If their exponents are equal, do they also have the same quantum?

35.1.4.(2)+ In Figure 439 on page 683, how many significant digits are there in each value?

35.1.5.(2)+ In Figure 439 on page 683, how many members are there in the cohort of each value?

35.1.6.(3)+ Suppose a decimal floating-point representation supports p significand digits. If a nonzero value has n digits between its leftmost and rightmost nonzero digits, how many members are in its cohort, assuming the value is well within the limits of the representation's exponent range? (For example, if p=11 and the significand is 00057036000, then n=5.)

35.2. System z Decimal Floating-Point Data Encoding and Representation (*)

Details Follow

This next section describes the interesting (and complex!) details of the System z decimal floating-point format, but you don't need to understand it to make good use of decimal floating-point instructions and arithmetic. The conceptual representation in Figure 438 on page 682 is an adequate description for almost all your needs.

We'll start by showing how three BCD digits are encoded in 10 bits.

35.2.1. Decimal Floating-Point Data Encoding (*)

To efficiently encode decimal values between 0 and 999, we could use 10-bit binary integers. (The values from 1000 to 1023 would be unused, but this is a small price to pay.) However, because we want to do *decimal* arithmetic, we should avoid frequent internal conversions between binary and decimal. Thus, this 10-bit binary-integer format was rejected.

Instead, an efficient and unusual *densely packed decimal* (DPD) encoding is used. Three BCD digits (12 bits) are mapped into an unusual 10-bit grouping[229] called a “declet”.

The actual encoding takes this form: let 3 BCD digits be represented by the 12 bits “abcd efgh ijkm”, and let a a 10-bit declet be represented by “pqr stu v wxy”. Converting from BCD digits to a declet is done in two steps:

1. Extract the high-order bits of the three BCD digits (aei).
2. Select one of eight 10-bit declet encodings, based on the remaining nine bits of the BCD string. Table 320 shows the bit selections:

aei	pqr stu v wxy
000	bcd fgh 0 jkm
001	bcd fgh 1 00m
010	bcd jkh 1 01m
011	bcd 10h 1 11m
100	jkd fgh 1 10m
101	fgd 01h 1 11m
110	jkd 00h 1 11m
111	00d 11h 1 11m

Table 320. Declet encoding for BCD digits

For example, suppose you want to encode the BCD digits 579 into a declet. The bits of the BCD digits are 0101 0111 1001, so the aei bits are 001. The remaining three bits (bcd) of the first BCD digit are 101; the remaining three bits of the second BCD digit (fgh) are 111, and the remaining bits (jkm) of the last BCD digit are 001. When combined, the result is 101 111 1 001. (The jk bits are zero because the final BCD digit is 9, so they are ignored in forming the declet!)

Because we are encoding only 1000 values (from 000 to 999), the remaining 24 possible bit patterns are called “non-preferred”, and assigned 3 each to 8 BCD values. They have bit patterns 01x11x111x, 10x11x111x, and 11x11x111x, where x can be either 0 or 1. The preferred encodings are called *canonical encodings*. Non-preferred encodings are accepted as arithmetic operands, but are never generated as a result.

Figure 469 on page 734 shows all the BCD-to-DPD encodings, including the 24 non-preferred declet encodings. Figure 470 on page 735 shows all the DPD-to-BCD encodings, and Table 386 on page 733 summarizes the non-canonical declets and their BCD equivalents.

Declets can be converted to BCD digits according the rules shown in Table 321 on page 686, where the '-' is a “don't care” indicator meaning the bit may be 0 or 1. (The same notation for declet bits and BCD-digit bits is used as in Table 320.)

[229] The details are ugly or elegant, depending on your perspective.

vwxst	abcd	efgh	ijkm
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

Table 321. Converting decimal floating-point declets to BCD digits

For example, suppose you start with a declet 1011111001, where the bits are denoted pqrstuvwxy, and the bits of the three BCD digits are denoted abcd efgh ijkm. The vwxst bits are then 10011. This corresponds to the second row of Table 321, so we construct the abcd (first BCD) digit from 0101, the efgh (second BCD) digit from 0111, and the ijkm (third BCD) digit from 1001. Thus, the three BCD digits are 579, as desired.

This may seem an enormous amount of effort and a great complication, just to encode three BCD digits. But it has a great advantage from the perspective of the CPU designers: DPD-to-BCD and BCD-to-DPD conversions can be done efficiently with simple logical operations;[230] no arithmetic is needed!

Remember that all the non-preferred digit encodings have bit representations 01x11x111x, 10x11x111x, and 11x11x111x, where x is either 0 or 1.

35.2.2. Decimal Floating-Point Data Representation (*)

Now that we know how declets are created, we'll see how they and the exponent are actually packaged. Unlike Figure 438 on page 682, the decimal floating-point representation takes the form shown in Figure 440:

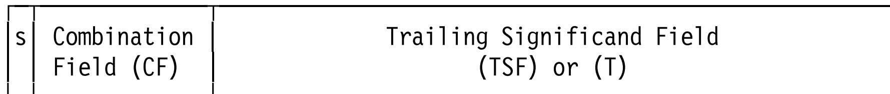


Figure 440. Decimal floating-point data representation

The key properties of the representation shown in Figure 440 are:

- The coefficient is a string of encoded decimal digits, to the left of an implied decimal point following the low-order digit. The value of the exponent determines the true position of the decimal point (as we've seen for hexadecimal and binary floating-point data).
- The rightmost digit is the “units digit”. It may be zero, because the significand is *not* right-normalized.
- The leftmost nonzero digit is called the “leftmost significant digit”.
- The high-order digit (whether or not it's zero) is the “leftmost digit”, Thus, the *significand* digits are the leftmost digit through the low-order units digit, while the *significant* digits are the leftmost *nonzero* digit through the low-order units digit.

Then, the value of a number is $(-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}$

Figure 441 on page 687 shows the 32-, 64-, and 128-bit formats. Each Trailing Significand Field (sometimes abbreviated TSF) is a multiple of 10 bits long, because it contains 2, 5, or 11 declets.

[230] The logical operations are described in the IEEE Standard.

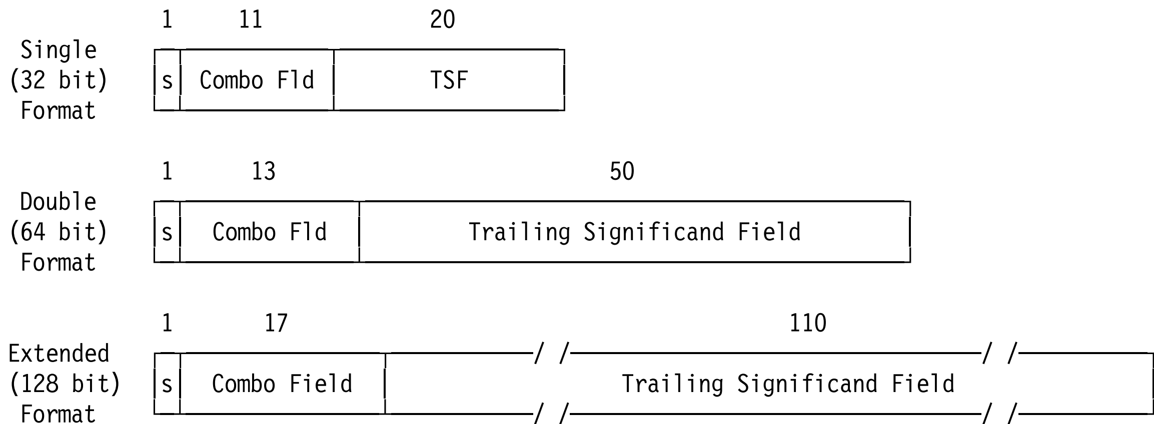


Figure 441. System z decimal floating-point representations

The 32-bit format is not encouraged by the IEEE standard, but is supported by the Assembler as a storage format. There are no arithmetic operations on the 32-bit format, but special instructions make it easy to convert to and from long and extended formats.

The Combination Field is unusual: it contains the exponent *and* the leftmost significand digit! The Trailing Significand Field (TSF) contains the remaining significand digits, encoded as 10-bit declets. Thus, the precisions of the three representations are 6(+1)=7, 15(+1)=16, and 33(+1)=34 decimal digits respectively.

35.2.3. Decimal Floating-Point Combination Field (*)

The representation of the exponent is unusual: the first five bits of the Combination Field[231] (CF) contain the leftmost significand digit and the first two bits of the characteristic (the biased exponent), and the remaining 6, 8, or 12 bits contain the rest of the characteristic.

The first five bits allow 32 combinations; two are reserved for the special values shown in Table 322:

CF bits 0-4	Meaning
11111	NaN
11110	Infinity
All others	Finite numbers

Table 322. First five bits of special-values Combination Field

When the first 5 bits are 11111, the data item is a NaN. If the *next* bit (bit 6 of the representation; remember, the sign bit is bit 0) is zero, the item is a QNaN, and if one, a SNaN. Canonical NaNs set all other bits of the data item to zero.

When the first 5 bits are 11110, the data item is an infinity, no matter what the remaining bits are. Canonical infinity values set all the other bits of the data item to zero.

The remaining 30 possible combinations of Combination Field are used to encode finite values. The five bits contain the Leftmost Digit (LMD) of the significand *and* the first *two* bits of the characteristic, as shown in Table 323 on page 688.

[231] The IEEE 754-2008 standard calls these bits G_0 through G_4.

LMD	First 2 characteristic bits		
	00	**01**	**10**
0	00000	01000	10000
1	00001	01001	10001
2	00010	01010	10010
3	00011	01011	10011
4	00100	01100	10100
5	00101	01101	10101
6	00110	01110	10110
7	00111	01111	10111
8	11000	11010	11100
9	11001	11011	11101

Table 323. First 5 bits of finite-value Combination Field

A zero value has all significand digits zero, and any representable exponent.

Suppose we want to represent the value 8765432 as a short decimal floating-point number. The rightmost 6 digits are encoded into two declets `X'3E5'` (representing 765) and `X'232'` (representing 432). Thus, the rightmost five hex digits of the data item contain `X'F9632'`. (Remember, the declet representing 765 was shifted right 2 bits.)

The exponent is zero, so the characteristic is decimal 101 (the characteristic bias, shown in Table 324 on page 689) or `B0110 0101'`. Because the Combination Field is 11 bits wide and its first 5 bits are the LMD+Leading CF bits, we must extract the two leading bits of the characteristic, or 01. Then, we see in Table 323 that the LMD 8 means that the leading 5 bits of the CF contains 11010. With a zero for the sign bit, we can put the pieces together:

- `0` (Sign Bit)
- `11010` (initial 5 bits of Combination Field)
- `100101` (rest of the characteristic)
- `X'F9632'` (20-bit trailing significand)

so the final data item has representation `X'6A5F9632'`.

Decimal floating-point encodings are difficult!

As this example shows, it's best to let the Assembler and the CPU generate encoded values for you.

Table 324 on page 689 summarizes properties of the decimal floating-point representation. The values in the row denoted “Characteristic length” show that two of its bits are encoded in the Combination Field.

Property	Short format	Long format	Extended format
Format length (bits)	32	64	128
Combination field length (bits)	11	13	17
Characteristic length (bits)	6+2	8+2	12+2
Trailing Significand Field length (bits)	20	50	110
Precision (decimal digits)	7	16	34
Exponent range	−95, +96	−383, +384	−6143, +6144
Exponent bias	101	398	6176
Max Normalized Value (N_{max})	$\approx 10^{+97}$	$\approx 10^{+385}$	$\approx 10^{+6145}$
Min Normalized Value (N_{min})	$\approx 10^{-95}$	$\approx 10^{-383}$	$\approx 10^{-6143}$
Smallest DeNorm Value (D_{min})	$\approx 10^{-101}$	$\approx 10^{-398}$	$\approx 10^{-6176}$
Effective exponent range (of all representable values)	−101, +90	−398, +369	−6176, +6111

Table 324. Properties of decimal floating-point representations

The unusual format of the combination field allows the exponent range to be wider: normally an n-bit exponent would allow 2^n values, but the DFP encoding allows approximately $3\times2^{(n-1)}$ or 1.5 times as many values.

Exercises

35.2.1.(2)+ Using a decimal floating-point notation like that used in the text on page 683, what are the quanta of each member of the cohorts of these numbers?

1. [−3‖7430]
2. [+1‖0009]
3. [−2‖0340]

35.2.2.(1) In Figure 441 on page 687, how many declets appear in each of the decimal floating-point representations?

35.2.3.(2) Suppose the significand of a long decimal floating-point number has eight nonzero significant digits, such as “12345678”. How many members does its cohort have?

35.2.4.(3) Suppose the significand of a long decimal floating-point number has eight significant digits, but you don't know their values. How many members does its cohort have?

35.2.5.(2) What is the declet encoding of the BCD digits 987? Show the steps you used to derive your result.

35.2.6.(2) Decode the declet 1011001101 to BCD digits. Show the steps you used to derive your result.

35.2.7.(2) What type of long decimal floating-point special value is represented by these strings?

1. `X'827C000000000000'`
2. `X'7EF00049826BA3B0'`
3. `X'FB7388215142D357'`

35.2.8.(1) Write and assemble a test program to verify that the representation of the short decimal floating-point value 8765432 is as derived following Table 323 on page 688.

35.2.9.(2)+ Convert the value 3.141593 to short and long decimal floating-point using the method shown following Table 323 on page 688.

35.2.10.(2) Why is the low-order digit of a decimal floating-point value often the same as the low-order digit of its decimal value?

35.3. Decimal Floating-Point Constants

Decimal floating-point constants are generated by DC statements with types E, D, and L for short, long, and extended formats respectively. The type extension must be D:

ED Short format decimal floating-point

```
2614D2E7            DC  ED'123.4567'      7 digits
```

DD Long format decimal floating-point

```
261934B9C1E28E56    DC  DD'12345678.90123456'     16 digits
```

LD Extended format decimal floating-point

```
2603534B9C1E28E5    DC  LD'123456789012345.6789012345678901234'    34 digits
6F3C127177823534
```

These three statements generate the short, long, and extended decimal floating-point representations of 100:

```
DC   ED'100'            Generates X'22500080'
DC   DD'100'            Generates X'2238000000000080'
DC   LD'100'            Generates X'2208000000000000 0000000000000080'
```

Because decimal floating-point values are not normalized, equivalent values (members of a cohort) can generate different constants. For example, each of these constants has numeric value 100;

```
DC   ED'100.00'         Generates X'22304000'
DC   ED'100.0'          Generates X'22400400'
DC   ED'100'            Generates X'22500080'
DC   ED'1E2'            Generates X'22700001'
```

but the generated constants depend on the number of significant digits in the nominal value: in the first three constants, the trailing zero digits are significant.

You can also generate constants with the four special values infinity, NaN, SNaN, and QNaN, as well as the three finite values Max, Min, and DMin. These are shown in Table 325:

Value	Short	Long	Extended
(Inf)	78000000	7800000000000000	7800000000000000 0000000000000000
(NaN)	7C000000	7C00000000000000	7C00000000000000 0000000000000000
(SNaN)	7E000000	7E00000000000000	7E00000000000000 0000000000000000
(QNaN)	7C000000	7C00000000000000	7C00000000000000 0000000000000000
(Max)	77F3FCFF	77FCFF3FCFF3FCFF	77FFCFF3FCFF3FCF F3FCFF3FCFF3FCFF
(Min)	04000000	0400000000000000	0400000000000000 0000000000000000
(DMin)	00000001	0000000000000001	0000000000000000 0000000000000001

Table 325. Assembled decimal floating-point special-value constants

Unlike binary floating-point, the only decimal floating-point special value that is not simply a short constant extended with zeros is the (Max) constant. (Compare these values with those in Table 276 on page 644.)

Decimal floating-point zeros do *not* have a unique representation, as illustrated in Table 326 on page 691. This lack of uniqueness can have possibly unexpected effects in decimal floating-point arithmetic operations; some examples are given in Section 36.7.

DC Operand	Generated Constant
ED'0'	X'22500000'
ED'0.0'	X'22400000'
ED'.00'	X'22300000'
ED'0E-101'	X'00000000'
DD'0'	X'22380000 00000000'
DD'0E1'	X'223C0000 00000000'
DD'0E369'	X'43FC0000 00000000'
DD'0E-398'	X'00000000 00000000'

Table 326. Examples of decimal floating-point short precision zeros

Unlike hexadecimal and binary floating-point zeros, decimal floating-point zeros can have a wide variety of representations.

35.3.1. Rounding-Mode Suffixes for Decimal Floating-Point Constants

In the absence of a specific rounding request, the Assembler rounds decimal floating-point constants to "nearest even". For directed rounding, you can specify a rounding-mode suffix of the form **Rn**, where **n** is one of the numbers 8 through 15; **Rn** must given as a single token with no embedded blanks. Table 327 lists the rounding-mode suffixes and their meanings.

Mode	Rn	Description
8	R8	Round to nearest; ties to nearest even ("half-even")
9	R9	Round toward zero (truncate)
10	R10	Round to $+\infty$; if $-$, truncate
11	R11	Round to $-\infty$; if +, truncate
12	R12	Round to nearest; ties away from 0 ("half-up")
13	R13	Round to nearest; ties toward 0 ("half-down")
14	R14	Round up (away from zero)
15	R15	Round for reround, or "prepare for shorter precision"

Table 327. Assembler rounding-mode suffixes for DFP constants

We'll discuss these various rounding options in more detail later.

In this short-precision example, the nominal value is rounded up:

```
2614D2E7            DC  ED'123.4567'      Default rounding
2614D2E8            DC  ED'123.45678R10'  Rounded to +infinity
2614D2E8            DC  ED'123.45678R14'  Rounded away from zero
```

In this long-precision example, the nominal value is rounded down:

```
261934B9C1E28E58    DC  DD'12345678.901234577'    Default rounding
261934B9C1E28E57    DC  DD'12345678.901234577R11' Rounded to -infinity
```

In this extended-precision example, the nominal value is rounded toward zero (truncated):

```
2603534B9C1E28E5    DC  LD'123456789012345.67890123456789012349'   Default rounding
6F3C127177823535
2603534B9C1E28E5    DC  LD'123456789012345.67890123456789012349R9' Rounded to 0
6F3C127177823534
```

35.3.2. Decimal Exponents and Modifiers

You can specify a decimal exponent as part of the nominal value, or an exponent modifier that applies to all the nominal values in the operand. Figure 442 on page 692 shows some examples; the letters E, M, and B in the comments fields mean the constant has a decimal **E**xponent, an exponent **M**odifier, or **B**oth. The “also” comments show different ways to generate the same constant. All the constants have value 100, so are members of the same cohort.

```
DC    ED'10000E-2'      E: generates X'22304000'
DC    EDE-2'10000'      M: generates X'22304000' also.
DC    ED'1000E-1'       E: generates X'22400400'
DC    EDE-1'1000'       M: generates X'22400400' also.
DC    ED'100'              Generates X'22500080'
DC    ED'10E1'          E: generates X'22600010'
DC    ED'1E2'           E: generates X'22700001'
DC    EDE1'1E1'         B: generates X'22700001' also,
DC    EDE-5'1E7'        B: generates X'22700001' also, and
DC    ED'.1E3'          M: generates X'22700001' also.
```

Figure 442. DFP constants with exponent modifiers and decimal exponents

The only other modifier allowed with decimal floating-point constants is Length, and it must be 4 for short precision values, 8 for long, or 16 for extended. Its only effect is that the generated constants are not aligned on their respective “natural” boundaries: word, doubleword, and doubleword.

Exercises

35.3.1.(2)+ A four-byte area of memory contains the bit pattern X'4040405C'. What possibilities are represented by that pattern?

35.3.2.(3) Write three decimal floating-point constants that generate a value one “ulp” smaller than the (Max) constant in short, long, and extended representations.

35.3.3.(2) Write three decimal floating-point constants with numeric operands that generate a value equal to that generated by a (Max) operand.

35.3.4.(2)+ Hexadecimal and binary floating-point constants allow a variety of length modifiers. Why do decimal floating-point constants not allow length modifiers other than 4 (for short), 8 (for long), or 16 (for extended)?

35.3.5.(2)+ How many members are there in the cohort of +0 in a short decimal floating-point representation?

35.3.6.(2)+ How many members are there in the cohort of +0 in a long decimal floating-point representation?

35.3.7.(2)+ Assemble the following sixteen short DFP constants, and study the generated values, particularly the encoding of the significands. Which of the values are “NMin”, and which are “DMin”, having the minimum exponent?

```
10000000.E-95
 1000000.E-95
  100000.E-95
   10000.E-95
    1000.E-95
     100.E-95
      10.E-95
       1.E-95
       .1E-95
       .01E-95
       .001E-95
```

```
.0001E-95
.00001E-95
.000001E-95
.0000001E-95
.00000001E-95
```

35.4. Decimal Floating-Point Data Classes (*)

Like binary floating-point data types, decimal floating-point data occurs in six classes:

1. Zeros may have any exponent and either sign, and have a zero significand.[232] ±0's are distinct, but compare equal.
2. Subnormal numbers X lie in the range $D_{min} \le |X| < N_{min}$
3. Normal values X lie in the range $N_{min} \le |X| \le N_{max}$
4. Infinity requires that the first five bits of the Combination Field be 11110. The contents of the Trailing Significand Field is ignored. Infinities are valid in decimal floating-point arithmetic operations.
5. Quiet NaNs require that the first five bits of the Combination Field be 11111, and that the next bit of the CF is 0.
6. Signaling NaNs require that the first five bits of the Combination Field be 11111, and that the next bit of the CF is 1.
   - Operations involving SNaNs usually cause an Invalid Operation exception, and may be masked off by setting mask bits in the Floating-Point Control Register (described in Section 34.4 on page 649).

Table 328 lists the three instructions used to test operands in the floating-point registers:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED50	TDCET	RXE	Test Data Class (Short)	ED54	TDCDT	RXE	Test Data Class (Long)
ED58	TDCXT	RXE	Test Data Class (Extended)				

Table 328. Decimal floating-point Test Data Class instructions

The operands of these instructions have the format

$R_1,D_2(X_2,B_2)$

where R_1 designates a FPR.

A bit pattern in the rightmost 12 bits of the Effective Address of the second operand address tests selected classes of the first operand; the remaining bits of the Effective Address are ignored. The instruction sets the CC to 1 if a bit in the Effective Address matches the operand's data class, or to zero if not.

Table 329 on page 694 shows the test bits; the bit numbering corresponds to a 64-bit Effective Address.

[232] Binary floating-point zeros must have zero characteristic and zero significand. Hexadecimal floating-point true zeros have zero characteristic and zero fraction, but pseudo-zeros have a zero fraction and *nonzero* characteristic.

Class	+ sign	− sign
Zero	52	53
Subnormal	54	55
Normal	56	57
Infinity	58	59
QNaN	60	61
SNaN	62	63

Table 329. DFP Test Data Class second-operand bits

Note:

The order of the test bits in the second operand's Effective Address of these decimal floating-point instructions is different from the order of the test bits in the corresponding binary floating-point instructions. (Compare Table 329 to Table 287 on page 654.)

To illustrate, the Condition Code after each of these TDCET instructions in Figure 443 will be 1, because the bits in the Effective Address match the class of the data in FPR0.

```
LZER  0                  c(FPR0) = X'00000000', zero
TDCET 0,B'110000000000'  Test for zero
LE    0,=ED'1E-100'      c(FPR0) = X'00100001', subnormal
TDCET 0,B'001100000000'  Test for subnormal
LE    0,=ED'1'           c(FPR0) = X'22500001', normal
TDCET 0,B'000011000000'  Test for normal
LE    0,=ED'(Inf)'       c(FPR0) = X'78000000', infinity
TDCET 0,B'000000110000'  Test for infinity
LE    0,=ED'(QNaN)'      c(FPR0) = X'7C000000', QNaN
TDCET 0,B'000000001100'  Test for QNaN
LE    0,=ED'(SNaN)'      c(FPR0) = X'7E000000', SNaN
TDCET 0,B'000000000011'  Test for SNaN
```

Figure 443. Examples of decimal floating-point Test Data Class instructions

Table 330 shows another way to visualize the left-to-right order of the test bits and the tested classes:

Bits 0-51	52	53	54	55	56	57	58	59	60	61	62	63
Ignored	+Zero	−Zero	+Subnormal	−Subnormal	+Normal	−Normal	+Infinity	−Infinity	+QNaN	−QNaN	+SNaN	−SNaN

Table 330. Test Data Class test-bit vs. tested-class correspondence

Three related “Test Data Group” instructions are described in Section 35.12.7 on page 726.

Exercises

35.4.1.(2) Given a decimal floating-point operand in FPR0, which data classes will these instructions detect?

```
(1)       TDCET 0,29
(2)       TDCDT 0,4095
(3)       TDCXT 0,15
(4)       TDCDT 0,X'C00'
```

35.5. Decimal Floating-Point Operations: Rounding, Quanta, and Exceptions

Decimal floating-point operations offer rich choices of rounding options; we'll describe them briefly. Because many instructions involve selecting the "quantum" of the final result, we'll see some examples to clarify the concepts involved.

35.5.1. Rounding

Rounding a result requires choosing a "Permissible Value" among the possible values; it is usually one of the values allowed by the destination precision. (Infinity is not a permissible value.)

Suppose Z is the infinitely precise result.

- If Z is exactly representable, its value is always selected as the delivered result, independent of the rounding mode.
- Otherwise, the two adjacent values with same sign as Z are candidates for the delivered result.

To illustrate, suppose Z is the intermediate result of an operation, as sketched in Figure 444.

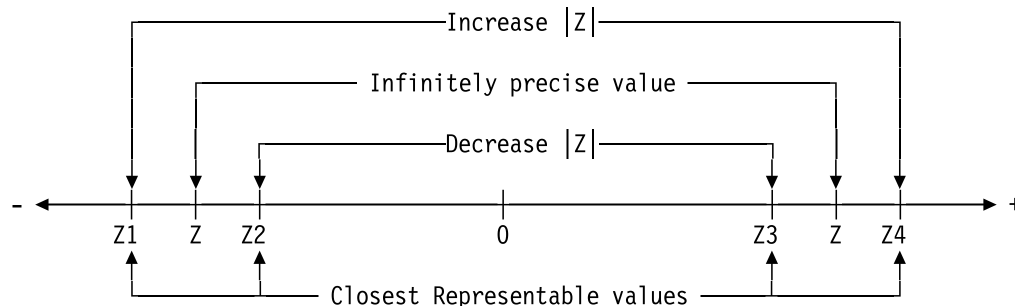


Figure 444. Illustration of decimal floating-point rounding candidates

Let Z1 and Z2, or Z3 and Z4 be the nearest representable values to the infinitely precise value. If Z is not representable, either Z1 or Z2 is chosen (if Z is negative), or Z3 or Z4 is chosen (if Z is positive). Z2 or Z3 can be zero.

Assuming Z is not between Z2 and Z3, the standard rounding modes from Z to one of the nearest candidate values are as follows:

- If Z lies exactly halfway between the two candidates, a "tie" occurs: choose the candidate with an even low-order digit. ("Round to nearest even")
- If Z lies exactly halfway between the two candidates, a "tie" occurs: choose the candidate with the larger magnitude (Z1 or Z4). ("Round half-up")
- Choose the candidate toward $+\infty$ (Z2 or Z4). ("Round up")
- Choose the candidate toward $-\infty$ (Z1 or Z3). ("Round down")
- Choose the candidate closer to zero (Z2 or Z3). ("Round toward zero; Truncate")
- If Z lies anywhere between the two candidates, choose the candidate with the larger magnitude (Z1 or Z4). ("Round away from zero")

If Z is a very small number close to zero, there are two cases to consider:

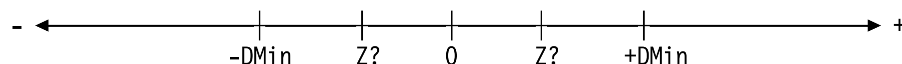


Figure 445. Illustration of decimal floating-point rounding candidates near zero

- If -DMin < Z < 0, then
  - -DMin is selected for for rounding toward $-\infty$ or for rounding away from zero

– Otherwise, 0 is selected.

- If 0 < Z < +DMin, then
  - +DMin is selected for for rounding toward +∞ or for rounding away from zero
  - Otherwise, 0 is selected.

These rounding modes are illustrated in Table 331:

Rounding Mode	12.341	12.345	12.349	12.355	–12.345
Round to nearest, ties to even	12.34	12.34	12.35	12.36	–12.34
Round to nearest, ties away from zero	12.34	12.35	12.35	12.36	–12.35
Round toward +∞	12.35	12.35	12.35	12.36	–12.34
Round toward –∞	12.34	12.34	12.34	12.35	–12.35
Round toward zero (truncate)	12.34	12.34	12.34	12.35	–12.34
Round away from zero	12.35	12.35	12.35	12.36	–12.35

Table 331. Example of DFP rounding modes

We'll see how to set a default "global" decimal floating-point rounding mode in Section 35.12 starting on page 718; one other rounding type ("Round to prepare for shorter precision") is described there.

35.5.2. Preferred Exponent and Quantum

When a decimal floating-point calculation generates a result, if the result is inexact to the operands' precisions, a choice must sometimes be made among several possible cohort members having the correct value. Using numbers with seven-digit maximum precision, suppose we add two numbers having the same exponent (and quantum):

```
 123.450           123450.×10**-3  Exponent = -3, Quantum = .001
+  6.228             6228.×10**-3  Exponent = -3, Quantum = .001
 129.678           129628.×10**-3  Exponent = -3, Quantum = .001
```

the sum has the same exponent and quantum so long as greater precision isn't needed for the significand.

Suppose the two numbers have different exponents/quanta:

```
 123.45             12345.×10**-2  Exponent = -2, Quantum = .01
+  6.228             6228.×10**-3  Exponent = -3, Quantum = .001
 129.678           129628.×10**-3  Exponent = -3, Quantum = .001
```

In this case, the same result should have the same quantum as before, so we assign the smaller exponent/quantum of the two operands. This choice preserves all the significant digits.

It doesn't matter if the low-order digit is zero; the smaller quantum is still chosen: this is the "preferred quantum" for exact results.

```
 123.45             12345.×10**-2  Exponent = -2, Quantum = .01
+  6.220             6220.×10**-3  Exponent = -3, Quantum = .001
 129.670           129670.×10**-3  Exponent = -3, Quantum = .001
```

If the result is inexact, a different rule is needed. Using 7-digit precision, suppose we add:

```
 123.45             12345.×10**-2  Exponent = -2, Quantum = .01
+  6.22801         622801.×10**-5  Exponent = -5, Quantum = .00001
 129.67801                        Quantum = ???, inexact result
```

The result will be inexact because it must be rounded to seven significant digits (the example uses rounding toward zero). Thus, the possible same-valued results with 7 digits are 129.6780 and 0129.678; these are two members of a cohort having quanta .0001 and .001 respectively. The

smaller quantum is preferred, so the first value 129.6780 is delivered; it is the result with the greater precision.

A similar rule applies to multiplication: the exponent of the product is the sum of the two operand exponents, and the product quantum is the product of the two operand quanta. Assuming 7-digit precision,

```
   11.11             1111.×10**-2  Exponent = -2, Quantum = .01
 × 22.2               222.×10**-1  Exponent = -1, Quantum = .1
 246.642           246642.×10**-3  Exponent = -3, Quantum = .001, exact result
```

For an exact result, the exponent is the *sum* of the two exponents, and the quantum is the *product* of the quanta of the two operands. Now, suppose we generate the inexact product of 11.01 and 22.002, and the exact result 242.242020 must be rounded:

```
    11.01            1101.×10**-2  Exponent = -2, Quantum = .01
 ×  22.002          22002.×10**-3  Exponent = -3, Quantum = .001
  242.24202      24224202.×10**-5  Exponent = -5, Quantum = .00001, inexact
```

Again assuming the result is truncated to 7 significant digits, we must choose one of two possible cohort members: 242.2420 (with quantum .0001) and 0242.242 (with quantum .001). As we saw for addition, the preferred quantum is the smaller, so the delivered result is 242.2420, with the greater precision.

Similar considerations apply to division. The exponent of the quotient is the exponent of the divisor subtracted from the exponent of the dividend, and the quantum of the quotient is the quantum of the dividend divided by the quantum of the divisor. Consider an exact result:

```
   655.36           65536.×10**-2  Exponent = -2, Quantum = .01
 ÷  12.8              128.×10**-1  Exponent = -1, Quantum = .1
    51.2              512.×10**-1  Exponent = -1, Quantum = .1
```

For this exact result, the preferred quantum is the *quotient* of the operand quanta, in this case .01÷.1 = .1.

Now, consider an inexact result: the quotient of 10000.1 ÷ 2.0001 is 4999.8000099..., which must be rounded to seven significant digits:

```
     10000.1       100001.×10**-1  Exponent = -1, Quantum = .1
 ÷       2.0001     20001.×10**-4  Exponent = -4, Quantum = .0001
    4999.800...                    Exponent = -3, Quantum = .001
```

The cohort containing the result has three members: 004999.8 (with quantum 0.1) 04999.80 (with quantum .01), and 4999.800 (with quantum .001). The chosen result is the last of these, with the smallest exponent and quantum, and the largest number of significant digits.

This example shows an inexact quotient with only one cohort member:

```
       10.              0.×10**0   Exponent =  0, Quantum = 1
 ÷      .03             3.×10**-2  Exponent = -2, Quantum = .01
 333.3333...      3333333.×10**-4  Exponent = -4, Quantum = .0001
```

where the infinitely-repeating result has been truncated to 7 digits. If the result had been exact, the quantum would have been 100; but for an inexact result, the cohort member (in this case, there's only one!) with the smallest quantum is chosen.

These rules are summarized in Table 332 on page 698.

Operation	Result	Preferred Quantum
Add, Subtract	Exact	Smaller of the two source quanta, or the quantum closest to it
	Inexact	Smallest quantum of result's cohort members
Multiply	Exact	Product of the two source quanta
	Inexact	Smallest quantum of result's cohort members
Divide	Exact	Quotient of the two source quanta
	Inexact	Smallest quantum of result's cohort members

Table 332. Preferred quanta for some decimal floating-point operations

These examples help show that the exponent and quantum chosen for a delivered result generally retain the maximum number of significant digits. Low-order digits are preserved until the number of significant digits exceeds the precision p of the representation. Thus, values are right-normalized so long as the number of significant digits is less than p, and left-normalized when the number is greater than p.

We might characterize the treatment of preferred exponent and quantum[233] this way:

Preferred Exponent and Quantum

Decimal floating-point arithmetic tries to give the best available fixed-point result unless it can't, and then it gives you the most precise available rounded floating-point result.

35.5.3. DFP Exceptions

In addition to the exceptions described in Section 34.4 starting on page 649, the quantum exception is provided for decimal floating-point. This exception condition is caused when a result has a different quantum from the preferred quantum. This is rarely a concern to most programs.

There are two decimal floating-point “q” bits defined in the Floating-Point Control Register, shown in Figure 446. (The decimal floating-point rounding mode bits are described in Section 35.12.)

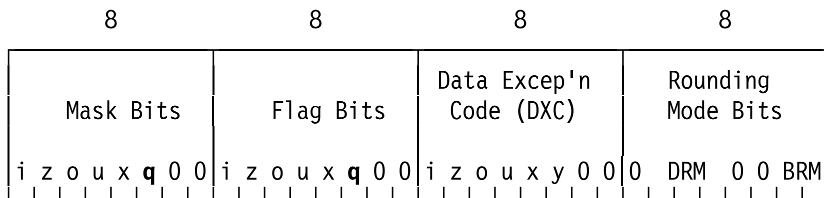


Figure 446. Floating-Point Control (FPC) register

Bit	Meaning
q (bit 5)	Quantum exception

In addition to the Data Exception Code values listed in Table 278 on page 650, there are two further codes that might be placed in the DXC when you execute a DFP instruction:

[233] Rather than the term “preferred quantum”, the IEEE-754 standard uses “preferred exponent” instead:

“For all computational operations except quantize and roundToIntegralExact, if the result is inexact the cohort member of least possible exponent is used to get the maximum number of significant digits. If the result is exact, the cohort member is selected based on the preferred exponent for a result of that operation, a function of the exponents of the inputs. Thus, for finite x, depending on the representation of zero, 0+x might result in a different member of x's cohort. If the result's cohort does not include a member with the preferred exponent, the member with the exponent closest to the preferred exponent is used.”

DXC	Meaning
03	a DFP instruction is attempting to execute on a CPU where DFP instructions are available, but are not enabled.
04	a DFP instruction has delivered a result with a quantum that is different from the preferred quantum.

Table 333. Decimal floating-point additional DXC value

The CPU's behavior for a quantum exception is shown in Table 334:

Mask bit = 1	Mask bit = 0
An interruption occurs: the DXC is set to X'04', and the quantum of the result is set to the quantum closest to the preferred quantum.	The result is the cohort member with quantum closest to the preferred quantum. The **q** flag bit is set to 1.

Table 334. Decimal floating-point quantum exception

In other words, the delivered result is the same whether or not you enable an exception.

The other exceptions generated by decimal floating-point operations are the same as those for binary floating-point described in Section 34.4.3 on page 651, and the other mask and flag bits in the first and second bytes are as described in Section 34.4 on page 649.

35.4.4. Overflow/Underflow Scale Factors (*)

When exponent overflow and underflow exceptions are enabled, the exponent of the delivered result is *scaled* by adding or subtracting a factor that causes the exponent to lie near the middle of the exponent range for that representation. These scale factors are shown in Table 335.

Long	Extended
576	9216

Table 335. Decimal floating-point scale factors for exponent spills

For overflows, the scale factor is subtracted from the overflowed exponent, and for underflows it is added to the underflowed exponent.

Exercises

35.5.1.(2)+ Does the choice of the smallest quantum for an inexact result mean that the significand is left-normalized?

35.5.2.(1) What will happen if you attempt to execute a DFP instruction on a CPU where DFP instructions are not available?

35.6. Decimal Floating-Point Data Movement Instructions

These instructions were described in Section 31.9 starting on page 568, and are mentioned here because they can be used for decimal floating-point operands. They have no dependence on any floating-point representation, and the CC is not changed.

35.6.1. Copy Sign

The CPSDR instruction copies the long second operand to the first operand register, giving it the sign of the third operand.

Op	Mnem	Type	Instruction
B372	CPSDR	RRF	Copy Sign (Long)

Table 336. Copy Sign instruction

The instruction format is

```
CPSDR R1,R3,R2
```

so you could write

```
CPSDR 1,5,8              Copy FPR8 to FPR1 with FPR5's sign
```

35.6.2. Copy between General and Floating-Point Registers

The LDGR and LGDR instructions copy long operands between a GPR and a FPR.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3C1	LDGR	RXY	Load FPR from GPR (Long)	B3CD	LGDR	RXY	Load GPR from FPR (Long)

Table 337. Instructions moving data between FPRs and GPRs

Their instruction format is

```
mnemonic R1,R2
```

To copy the contents of GG3 to FPR6, and from FPR2 to GG0, you could write

```
LDGR  6,3                Copy c(GG3) to FPR6
LGDR  0,2                Copy c(FPR2) to GG0
```

35.6.3. Copy Among Floating-Point Registers

Table 338 lists the instructions for moving floating-point operands among the floating-point registers. (They were introduced in System/360 for moving hexadecimal floating-point operands, but can just as well be used for binary and decimal operands.)

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
38	LER	RR	Load FPR from FPR (Short)	28	LDR	RR	Load FPR from FPR (Long)
B365	LXR	RRE	Load FPR from FPR (Extended)				

Table 338. Instructions copying data between FPRs

For example:

```
LER   5,8                Copy short c(FPR8) to FPR5
LDR   4,2                Copy long c(FPR4) to FPR2
LXR   1,12               Copy extended c(FPR12,FPR14) to (FPR1,FPR3)
```

Always remember that both operands of the LXR instruction must refer to valid FPR pairs.

Exercises

35.6.1.(1) What is the difference between these instructions?

```
         LDR   8,1
         LDR   10,3
and
         LXR   8,1
```

35.7. Decimal Floating-Point Arithmetic Instructions

We'll start with the basic arithmetic operations of multiplication, division, addition, and subtraction; the instructions are listed in Table 339. All the operations are rounded according to the current rounding mode in the FPCR.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3D0	MDTR	RRR	Multiply (Long)	B3D8	MXTR	RRR	Multiply (Extended)
B3D0	MDTRA	RRR	Multiply (Long)	B3D8	MXTRA	RRR	Multiply (Extended)
B3D1	DDTR	RRR	Divide (Long)	B3D9	DXTR	RRR	Divide (Extended)
B3D1	DDTRA	RRR	Divide (Long)	B3D9	DXTRA	RRR	Divide (Extended)
B3D2	ADTR	RRR	Add (Long)	B3DA	AXTR	RRR	Add (Extended)
B3D2	ADTRA	RRR	Add (Long)	B3DA	AXTRA	RRR	Add (Extended)
B3D3	SDTR	RRR	Subtract (Long)	B3DB	SXDR	RRR	Subtract (Extended)
B3D3	SDTRA	RRR	Subtract (Long)	B3DB	SXDRA	RRR	Subtract (Extended)

Table 339. Decimal floating-point basic arithmetic instructions

All decimal floating-point arithmetic instructions are register-register operations, unlike some hexadecimal and binary floating-point instructions that support register-storage operations. All can be "non-destructive" because neither source operand need be overwritten by the result.

The format of the instructions *without* an "A" suffix on the mnemonics is shown in Table 340:

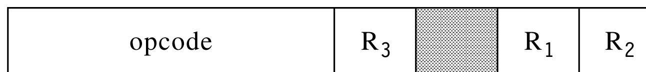


Table 340. Format of DFP arithmetic instructions

where the shaded field is filled with B'0000' by the Assembler.

Their assembler instruction statements have operand format

```
mnemonic R1,R2,R3
```

where the two source operands are in floating-point registers R_2 and R_3, and the result is placed in floating-point register R_1.

The operands must all be long or all extended precision; there are no arithmetic operations for short-precision or mixed-length operands.

The format of the instructions *with* an "A" suffix on the mnemonics is shown in Table 341:

Table 341. Format of DFP arithmetic instructions with rounding mask

allowing the programmer to specify a M_4 rounding mask value.

Their assembler instruction statements have operand format

```
mnemonic R1,M4,R2,R3
```

Again, the two source operands are in floating-point registers R_2 and R_3, and the result is placed in floating-point register R_1. The rounding mask M_4 applies *only* to the current instruction, and has the values and effects shown in Table 342 on page 702:

Mask	Effect	Mask	Effect
B'0000'	Use current DFP rounding mode	B'1000'	Round to nearest with ties to even
B'0001'	Round to nearest with ties away from 0 (“traditional” round)	B'1001'	Round toward 0
B'0010'	Use current DFP rounding mode	B'1010'	Round toward +∞
B'0011'	Round to prepare for shorter precision	B'1011'	Round toward −∞
B'0100'	Round to nearest with ties to even	B'1100'	Round to nearest with ties away from 0
B'0101'	Round toward 0	B'1101'	Round to nearest with ties toward 0
B'0110'	Round toward +∞	B'1110'	Round away from 0
B'0111'	Round toward −∞	B'1111'	Round to prepare for shorter precision

Table 342. Instruction-specific rounding mask values

The mnemonics with the “A” suffix have the same opcodes as the instructions without a suffix. The new mnemonics were introduced so the Assembler can correctly identify the four operands.

35.7.1. Multiplication

The decimal floating-point multiply instructions leave the Condition Code unchanged. You can think of the operation this way:

$$\text{operand1} \leftarrow \text{operand2} \times \text{operand3}$$

The exponent of the product is the sum of the exponents of the two operands.

Unlike hexadecimal and binary floating-point multiplication, there are no instructions to form “double-length” products from “single-length” operands; results have the same format as the operands. For example:

```
         LD    4,=DD'75.22'
         MDTR  1,4,4             c(FPR1) = c(FPR4)**2

         LD    8,=DD'1.41421'
         MDTR  2,4,8             c(FPR2)=75.22*1.41421

         LD    0,DDec1           High-order half to FPR0
         LD    2,DDec1+8         Low-order half to FPR2
         MXTR  8,0,0             c(FPR8,FPR10) = c(FPR0,FPR2)**2
         - - -
DDec1    DC    LD'0.1'           16-byte decimal floating-point constant
```

For finite results, the quantum of the product is chosen as shown in Table 332 on page 698.

If underflow or overflow occur, the appropriate exception is signaled; if enabled, the rounded result is delivered to the target register and an interruption occurs. Otherwise, the exponent is scaled and the corresponding flag bit in the FPCR is set to 1.

If either operand is a QNaN, that QNaN is the delivered result; if both are QNaNs, the R_2 operand is the result. Any SNaN operand, or multiplying zero and infinity, causes an invalid-operation exception.

To illustrate exponent scaling on overflow, suppose we multiply the decimal floating-point constant (Max) by itself, with the interruption enabled for exponent overflow:

```
LD    0,=DD'(Max)'       c(FPR0) = X'77FCFF3FCFF3FCFF'
MDTR  2,0,0              c(FPR2) = X'7500FF3FCFF3FCFE'
```

The overflowed exponent (384+384=768) is scaled by subtracting 576 (from Table 335 on page 699), giving a scaled exponent 192, which happens to be exactly half the maximum exponent, 384. The scaled result is now in “mid-range”.

35.7.2. Division

The decimal floating-point divide instructions leave the Condition Code unchanged, and no remainder — integer or fraction — is calculated. The second operand (the dividend) is divided by the third operand (the divisor). The quotient is rounded according to the current DFP rounding mode, and replaces the first operand. You can think of the operation this way:

```
operand1 ← operand2 ÷ operand3
```

The exponent of the quotient is the exponent of the divisor subtracted from the exponent of the dividend. For example:

```
LD    0,=DD'1'         c(FPR0) = 1   = X'2238000000000001'
LD    1,=DD'3'         c(FPR1) = 3   = X'2238000000000003'
DDTR  2,0,1            c(FPR2) = 1/3 = X'2DF9B36CDB36CDB3'
```

For finite results, the quantum of the quotient is chosen as shown in Table 332 on page 698.

If underflow or overflow occur, the appropriate exception is signaled; if enabled, the rounded result is delivered to the target register and an interruption occurs. Otherwise, the exponent is scaled and the corresponding flag bit in the FPCR is set to 1.

Dividing a finite number by infinity returns a zero quotient, and dividing a finite number by zero causes a divide-by-zero exception (which if masked off, returns infinity as the quotient).

Dividing zero by zero or infinity by infinity, or any SNaN operand, causes an invalid-operation exception. A SNaN operand is returned, unless both are SNaNs in which case the dividend SNaN is delivered. A QNaN divisor returns a canonical QNaN.

35.7.3. Addition and Subtraction

You can think of these operations this way:

```
operand1 ← operand2 ± operand3
```

The results from addition and subtraction depend on whether or not the result is exact. We'll illustrate using 7-digit operands:

- If the result is exact, the preferred quantum is the smaller of the quanta of the two source operands, or (if the smaller can't be represented) the quantum closest to the smaller.

  For example, this sum produces an exact result:

  ```
    1.234          Quantum = .001
  +  .34567        Quantum = .00001
    1.57967        Quantum = .00001
  ```

  and the result has the smaller quantum. In the next example, the result is exact but the smaller quantum can't be assigned:

  ```
    1.234000       Quantum = .000001 = 10**-6
  +  .3456700      Quantum = .0000001 = 10**-7
    1.5796700      Quantum = .0000001 ?
  ```

  In this case there are two cohort members with the same value: 1.579670 and 01.57967, with quanta 10^{-6} and 10^{-5} respectively. The quantum closest to the preferred value (10^{-7}) is 10^{-6}, so the delivered result is 1.579670.

- If the result is inexact, the preferred quantum is the smallest of the quanta of the result's cohort members.

For example, this sum produces an inexact result:

```
  1.234321          Quantum = .000001
+  .3454321         Quantum = .0000001
  1.579753          Quantum = .000001
```

and the (rounded) sum has the larger quantum because the cohort has only one member.

There may be more than one member of the cohort containing the result. For example:

```
  1.111201          Quantum = .000001
-  .1111009         Quantum = .0000001
  1.000100          Quantum = .000001
```

where the rounded result has discarded the low-order one. There are three cohort members: 1.000100, 01.00010, and 001.0001; the first has the smallest quantum and greatest number of significant digits among these cohort members.

These rules preserve as many low-order digits as possible.

Adding a finite number to infinity produces infinity, as does adding like-signed infinities. Adding infinities of opposite signs causes an invalid-operation exception. If the exception is masked off a default QNaN is delivered and the CC is set to 3.

Any SNaN operand delivers a SNaN result, and the CC is set to 3. QNaNs generate an invalid-operation exception; if masked off, the CC is set to 3.

For example:

```
LD    0,=DD'1'          c(FPR0) = 1
LD    4,=DD'(Inf)'      c(FPR4) = +infinity
ADTR  2,0,0             c(FPR2) = 2, CC=2
ADTR  2,0,4             c(FPR2) = +infinity, CC=2
SDTR  2,0,4             c(FPR2) = -infinity, CC=1
SDTR  2,4,4             Invalid-operation exception
```

Table 343 shows how the decimal floating-point add and subtract instructions set the Condition Code:

CC	Meaning
0	Result is zero
1	Result is less than zero
2	Result is greater than zero
3	Result is a NaN

Table 343. CC settings for Add/Subtract instructions

Exercises

35.7.1.(2)+ Assuming the same initial contents of FPR0 and FPR2 in each case, what result of each of these operations will be in FPR4?

```
         LD    0,=DD'-1'
         LD    2,=DD'-(Inf)'

(1)      DDTR  4,0,2
(2)      DDTR  4,2,0
(3)      MDTR  4,2,0
(4)      SDTR  4,0,2
```

35.8. Decimal Floating-Point Compare Instructions

We'll look at three groups of decimal floating-point compare instructions:

- arithmetic compare
- compare and signal
- compare biased exponent (characteristic)

All have operand format R_1,R_2. The first operand is compared to the second, and the CC is set to indicate the result, as shown in Table 344. All these compare instructions set the Condition Code in the same way.

CC	Meaning
0	Operands are equal
1	First operand is low
2	First operand is high
3	Operands are unordered

Table 344. CC settings for Compare instructions

35.8.1. Compare

These two instructions work just like other arithmetic comparisons, except that the presence of any QNaN sets the CC to 3. If either operand is a SNaN, an invalid operation exception occurs; if it is masked off, the CC is set to 3.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3E4	CDTR	RRE	Compare (Long)	B3EC	CXTR	RRE	Compare (Extended)

Table 345. Decimal floating-point Compare instructions

For example, these comparisons set the CC as indicated in the comments:

```
         LD    0,=DD'(Inf)'       c(FPR0) = +Infinity
         LD    1,=DD'(QNaN)'      c(FPR1) = +QNaN
         LD    2,=DD'2'           c(FPR0) = +2
         LD    3,=DD'(SNaN)'      c(FPR0) = +SNaN
*
         CDTR  0,2                CC = 2 (+Infinity > +2)
         CDTR  0,1                CC = 3 (unordered)
         CDTR  2,0                CC = 1 (+2 < +Infinity)
         CDTR  2,2                CC = 0
         CDTR  2,3                Invalid operation exception
```

35.8.2. Compare and Signal

The two Compare and Signal instructions in Table 346 behave just like the ordinary compare instructions, except that *any* NaN, Quiet *or* Signaling, causes an invalid operation exception. Again, if the exception is masked off, the CC is set to 3.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3E0	KDTR	RRE	Compare and Signal (Long)	B3E8	KXTR	RRE	Compare and Signal (Extended)

Table 346. Decimal floating-point Compare and Signal instructions

These comparisons set the CC as indicated in the comments:

```
         LD    0,=DD'(Inf)'        c(FPR0) = +Infinity
         LD    1,=DD'(QNaN)'       c(FPR1) = +QNaN
         LD    2,=DD'2'            c(FPR0) = +2
         LD    3,=DD'(SNaN)'       c(FPR0) = +SNaN
*
         KDTR  0,2                 CC = 2 (+Infinity > +2)
         KDTR  0,1                 Invalid operation exception
         KDTR  2,0                 CC = 1 (+2 < +Infinity)
         KDTR  2,2                 CC = 0
         KDTR  2,3                 Invalid operation exception
```

The main reason to use the Compare and Signal instructions is to avoid the propagation of NaNs through a computation, perhaps to stop before useless results have been generated.

35.8.3. Compare Biased Exponent

It is sometimes useful to compare the characteristics of two decimal floating-point operands using the instructions in Table 347.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F4	CEDTR	RRE	Compare Biased Exponent (Long)	B3FC	CEXTR	RRE	Compare Biased Exponent (Extended)

Table 347. Decimal floating-point Compare Biased Exponent instructions

These instructions compare the biased exponent (characteristic) of the operands, including infinity and any type of NaN. Table 348 shows the Condition Code setting:

CC	Meaning
0	Characteristics equal
1	First-operand characteristic is low
2	First-operand characteristic is high
3	Operands are unordered

Table 348. CC settings for Compare Biased Exponent instructions

If both operands are finite, the CC is set to 0, 1, or 2. Otherwise, comparing the characteristic of any infinity to another, or of any NaN to another, sets the CC to zero. Other combinations of operands set the CC to 3.

These examples show some biased exponent comparisons:

```
         LD    0,=DD'(Inf)'        c(FPR0) = +Infinity
         LD    1,=DD'(QNaN)'       c(FPR1) = +QNaN
         LD    2,=DD'2'            c(FPR0) = +2
         LD    3,=DD'2E75'         c(FPR0) = +2*(10**75)
*
         CEDTR 0,2                 CC = 3 (Finite vs. Infinity)
         CEDTR 1,1                 CC = 0 (NaN vs. NaN)
         CEDTR 2,1                 CC = 3 (Finite vs. NaN)
         CEDTR 2,3                 CC = 1 (2 vs. 2*(10**75))
         CEDTR 0,0                 CC = 0 (Infinity vs. Infinity)
```

Exercises

35.8.1.(1)+ Make a table showing the CC settings after comparing the biased exponents (characteristics) of a finite value, a NaN, and an infinity among one another.

35.9. Converting Decimal Floating-Point To and From Fixed Binary

The instructions in Table 349 convert fixed-point binary integers in a 64-bit general register to and from a decimal floating-point value in a floating-point register.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F1	CDGTR	RRE	Convert from Fixed (Long←64)	B3F1	CDGTRA	RRE	Convert from Fixed (Long←64)
B3F9	CXGTR	RRE	Convert from Fixed (Extended←64)	B3F9	CXGTRA	RRE	Convert from Fixed (Extended←64)
B3E1	CGDTR	RRF	Convert to Fixed (64←Long)	B3E1	CGDTRA	RRF	Convert to Fixed (64←Long)
B3E9	CGXTR	RRF	Convert to Fixed (64←Extended)	B3E9	CGXTRA	RRF	Convert to Fixed (64←Extended)

Table 349. Decimal floating-point convert to/from fixed binary instructions

35.9.1. Convert From Fixed Binary To DFP

These four instructions convert a 64-bit binary integer in the R_2 general register to a long or extended decimal floating-point value in the R_1 floating-point register, and the Condition Code is unchanged. For the instructions without an "A" suffix, the instruction format is

```
mnemonic R1,R2
```

For the instructions *with* an "A" suffix, the instruction format is

```
mnemonic R1,M3,R2,M4
```

The values of the M_3 rounding mask are shown in Table 342 on page 702, and the M_4 mask has bit settings to optionally suppress certain exception conditions.

The preferred quantum for an exact result is 1; for an inexact result, an exception might be indicated, and the preferred quantum is the smallest.

To illustrate, suppose we convert three binary values containing 17 decimal digits to long decimal floating-point, with default rounding:

```
LG    0,=FD'12345678901234560'     17 digits
CDGTR 0,0                         c(FPR0)=X'263D34B9C1E28E56'

LG    0,=FD'12345678901234565'     17 digits
CDGTR 0,0                         c(FPR0)=X'263D34B9C1E28E56'

LG    0,=FD'12345678901234569'     17 digits
CDGTR 0,0                         c(FPR0)=X'263D34B9C1E28E57'
```

In the first case, the result is exact because the final decimal digit is zero. The second case is rounded to the same value because the DFP value lies exactly between two decimal values so the result has an even low-order digit, and the third is rounded up to the next higher value.

Converting very large binary values does not cause overflow:

```
LG    0,=X'7FFFFFFFFFFFFFFF'  Maximum positive value
CDGTR 0,0                     c(FPR0)=X'6E45237C836973F6'
```

but because $2^{63}-1$ is 19 digits long, the decimal floating-point result was rounded to 16 decimal digits.

35.9.2. Convert From DFP To Fixed Binary

The instructions that convert from decimal floating-point to binary integers whose mnemonics have no suffix have the format shown in Table 350 on page 708:

opcode	M_3		R_1	R_2

Table 350. Format of Convert to Fixed Binary instructions

The assembler instruction statement format is

```
mnemonic R1,M3,R2
```

For the instructions with a suffix A, the instruction format is shown in Table 351:

opcode	M_3	M_4	R_1	R_2

Table 351. Format of Convert to Fixed Binary instructions

Their assembler instruction statement format is

```
mnemonic R1,M3,R2,M4
```

As noted above, the M_4 field can be used to mask certain exception conditions.

The decimal floating-point operand in the R_2 floating-point register is rounded to a binary integer in the R_1 general register according to the M_3 rounding modifier described in Table 342 on page 702.

Figure 447 shows some examples of proper conversions; those with CGXTR instructions use 16-byte extended-precision values that can hold all the digits of the full 19-digit maximum-integer value.

```
LD    0,=DD'127'
CGDTR 0,B'0000',0                  c(GG0)=X'000000000000007F'
LD    0,=DD'-3'
CGDTR 0,B'0000',0                  c(GG0)=X'FFFFFFFFFFFFFFFD'

LD    0,=LD'9223372036854775807'      2**63-1 = +MaxPos
LD    2,=LD'9223372036854775807'+8    2**63-1 = +MaxPos
CGXTR 0,B'0000',0                  c(GG0)=X'7FFFFFFFFFFFFFFF'

LD    0,=LD'-9223372036854775808'     -2**63 = -MaxNeg
LD    2,=LD'-9223372036854775808'+8   -2**63 = -MaxNeg
CGXTR 0,B'0000',0                  c(GG0)=X'8000000000000000'
```

Figure 447. Examples of converting decimal floating-point to fixed binary

In each case, the Condition Code is set according to the result, as shown in Table 352:

CC	Meaning
0	Source was zero
1	Source was less than zero
2	Source was greater than zero
3	Special case

Table 352. CC settings for Convert to Fixed instructions

Cases where the decimal floating-point value exceeds the representable range of a 64-bit binary integer are handled either by causing an invalid operation exception, or (if the exception is masked off) by setting the CC to 3 and the result to the maximum positive or negative integer, according to the sign of the decimal floating-point operand.

The following CGXTR instructions each cause an invalid operation exception, but because the exception is masked off the CC is set to 3 and the results are the default maximum and minimum integer values.

```
LFPC  =F'0'                             Mask off all exceptions
LD    0,=LD'9223372036854775808'    2**63 = +MaxPos+1
LD    2,=LD'9223372036854775808'+8  Low-order half
CGXTR 0,B'0000',0                   c(GG0)=X'7FFFFFFFFFFFFFFF'

LD    0,=LD'-9223372036854775809'   -2**63-1 = -MaxNeg-1
LD    2,=LD'-9223372036854775809'+8 Low-order half
CGXTR 0,B'0000',0                   c(GG0)=X'8000000000000000'
```

Figure 448. Examples of converting decimal floating-point to binary integer

Exercises

35.9.1.(3)+ Suppose two extended precision decimal floating-point numbers have the value of the 64-bit maximum positive integer plus 1/2 and the maximum negative integer minus 1/2:

```
LD    0,=LD'9223372036854775807.5'     +MaxPos+1/2
LD    2,=LD'9223372036854775807.5'+8   +MaxPos+1/2
CGXTR 0,Mp,0

LD    0,=LD'-9223372036854775808.5'    -MaxNeg-1/2
LD    2,=LD'-9223372036854775808.5'+8  -MaxNeg-1/2
CGXTR 0,Mm,0
```

For what values of the rounding modifier mask fields Mp and Mm will an invalid operation exception be avoided?

35.10. Converting Decimal Floating-Point To/From Packed and Zoned Decimal

As you might expect, converting decimal floating-point data to displayable decimal digits is far simpler than for hexadecimal and binary floating-point data. These instructions make the process quite simple. None of the instructions change the Condition Code.

The packed decimal operands are in general registers, not in memory.

35.10.1. Convert To/From Signed Packed Decimal

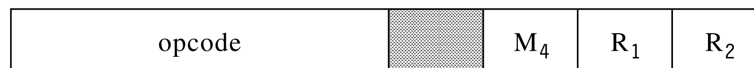
For signed packed decimal operands, use the instructions in Table 353:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F3	CDSTR	RRE	Convert from 64-bit Signed Packed (Long)	B3FB	CXSTR	RRE	Convert from 128-bit Signed Packed (Extended)
B3E3	CSDTR	RRF	Convert to 64-bit Signed Packed (Long)	B3EB	CSXTR	RRF	Convert to 128-bit Signed Packed (Extended)

Table 353. Decimal floating-point convert to/from signed packed decimal instructions

When converting *from* packed decimal to decimal floating-point you don't need to worry about rounding, because 15-digit and 31-digit packed decimal operands in a general register or general register pair (respectively) can be represented exactly in long and extended formats.

When you convert *to* packed decimal from decimal floating-point using the CSDTR and CSXTR instructions, you might want to select the sign code. the instruction format in Table 354 provides an M_4 field to select the sign code:



opcode		M_4	R_1	R_2

Table 354. Format of Convert to Signed Packed instructions

The M_4 mask uses only a single bit. If M_4=0, the + sign code of the packed decimal result is X'C', and if M_4=1, the + sign code is X'F'.[234] For negative numbers, the packed decimal sign code is X'D'.

The machine instruction statement operands for converting *to* packed decimal are written

```
mnemonic  R1,R2,M4
```

and those for converting *from* packed decimal are written

```
mnemonic  R1,R2
```

First, consider converting from signed packed decimal to decimal floating-point:

```
         LG    7,=PL8'123456789012345'      c(GG7)=X'123456789012345C'
         CDSTR 4,7                          c(FPR4)=X'2238A395BCF049C5'

         LG    8,=X'1234567890123456'       High-order 16 digits
         LG    9,=X'789012345678901C'       Low-order 15 digits, sign
         CXSTR 0,8                          Convert to extended DFP
*        c(FPR0,FPR2)=X'22080014D2E7078A 395BCF049C5DE08D'
```

Figure 449. Converting signed packed decimal to decimal floating-point

As this example indicates, both operands of the CXSTR instruction must be register pairs.

Converting from packed decimal to decimal floating-point is straightforward. Converting in the other direction, however, is a bit more complex. First, we'll start with the decimal floating-point values in Figure 449, and convert them to signed packed decimal.

```
         LD    4,=X'2238A395BCF049C5'  c(FPR4)=DD'123456789012345'
         CSDTR 7,4,B'0000'             c(GG7)=X'123456789012345C'

         LD    0,=X'22080014D2E7078A'  High-order half
         LD    2,=X'395BCF049C5DE08D'  Low-order half
         CXSTR 8,0,B'0000'             Convert to packed decimal
*        c(GG8,GG9)=X'1234567890123456 789012345678901C'
```

Figure 450. Converting decimal floating-point to signed packed decimal

In Figure 450 we started with decimal floating-point values containing 15 and 31 significant digits. Because long and extended decimal floating-point data can hold 16 and 34 significant digits, these two instructions will convert only the *rightmost* 15 and 31 digits to packed decimal, as shown in Figure 451.

```
         LD    0,=DD'1234567890123456' 16 significant digits
         CSDTR 3,0,B'0000'             c(GG3)=X'234567890123456C'

         LD    1,=LD'1234567890123456789012345678901234'   34 digits
         LD    3,=LD'1234567890123456789012345678901234'+8 34 digits
         CSXTR 4,1,B'0000'             Convert to packed decimal
*        c(GG4,GG5)=X'4567890123456789 012345678901234C'
```

Figure 451. Converting decimal floating-point to signed packed decimal

This example shows that one or three high-order digits can be lost when converting from decimal floating-point to *signed* packed decimal.

It doesn't matter if the DFP operand is a NaN or infinity; the “significand” is converted anyway. For example:

---

[234] Since M_4 is only a single bit, it's not really a “mask”; the other bit positions are reserved.

```
LD    0,=DD'(Inf)'       c(FPR0)=X'7800000000000000'
CSDTR 1,0,B'0000'         c(GG1)=X'000000000000000C'

LD    0,=DD'(SNaN)'      c(FPR0)=X'7E00000000000000'
CSDTR 2,0,B'0000'         c(GG2)=X'000000000000000C'
```

Be Careful!

When converting decimal floating-point data to signed packed decimal using the CSDTR instruction, you may lose one high-order digit; and when using CSXTR, you may lose three high-order digits.

We'll see in Section 35.12.4 that the Shift Significand instructions let you convert the “lost” digits.

35.10.2. Convert To/From Unsigned Packed Decimal

The instructions in Table 355 convert decimal floating-point data to *unsigned* packed decimal digits in a general register.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F2	CDUTR	RRE	Convert from 64-bit Unsigned Packed (Long)	B3FA	CXUTR	RRE	Convert from 128-bit Unsigned Packed (Extended)
B3E2	CUDTR	RRE	Convert to 64-bit Unsigned Packed (Long)	B3EA	CUXTR	RRE	Convert to 128-bit Unsigned Packed (Extended)

Table 355. Decimal floating-point convert to/from unsigned packed decimal instructions

Because no sign code is generated for the unsigned packed decimal results, 16 or 32 digits can be converted, and no digits are lost for long precision decimal floating-point data. CUXTR, however, loses the two high-order digits.

CUDTR and CUXTR provide no M_4 operand as do CSDTR and CSXTR, so their machine instruction statement operand format is R_1,R_2.

We'll use the examples as for conversions to signed packed decimal, to see the difference in the number of generated digits. Figure 452 shows how to convert from unsigned packed decimal to decimal floating-point.

```
         LG    7,=XL8'1234567890123456'     c(GG7)=X'1234567890123456'
         CDUTR 4,7                          c(FPR4)=X'263934B9C1E28E56'

         LG    8,=X'1234567890123456'       High-order 16 digits
         LG    9,=X'7890123456789012'       Low-order 16 digits
         CXUTR 0,8                          Convert to extended DFP
*        c(FPR0,FPR2)=X'2208012717782353 4B9C1E28E56F3C12'
```

Figure 452. Converting unsigned packed decimal to decimal floating-point

These examples convert from decimal floating-point to unsigned packed decimal:

```
         LD    4,=DD'1234567890123456'      c(FPR4)=X'263934B9C1E28E56'
         CUDTR 7,4                          c(GG0)=X'1234567890123456'

         LD    0,=LD'12345678901234567890123456789012'     High half
         LD    2,=LD'12345678901234567890123456789012'+8   Low half
         CUXTR 8,0                          Convert to packed decimal
*        c(GG8,GG9)=X'1234567890123456 7890123456789012'
```

Figure 453. Converting decimal floating-point to unsigned packed decimal

If we compare Figure 453 on page 711 to Figure 451 on page 710, we see that the lack of a sign code lets us generate one more packed decimal digit in place of the sign code.

Because it's easy to determine the sign of a decimal floating-point number, most programs converting decimal floating-point to packed decimal will probably use the unsigned conversions, to generate the largest possible number of digits.

NaNs and infinity are converted to unsigned packed decimal without causing exceptions:

```
LD    0,=DD'(Inf)'       c(FPR0)=X'7800000000000000'
CUDTR 1,0                 c(GG1)=X'0000000000000000'

LD    0,=DD'(SNaN)'      c(FPR0)=X'7E00000000000000'
CUDTR 2,0                 c(GG2)=X'0000000000000000'
```

Be Careful!

When converting decimal floating-point data to unsigned packed decimal using the CUXTR instruction, you may lose two high-order digits.

We'll see in Section 35.12.4 on page 720 that the Shift Significand instructions let you capture the “lost” digits.

35.10.3. Convert To/From Zoned Decimal

The instructions in Table 356 convert data between decimal floating-point and zoned decimal, saving the need to use packed decimal as an intermediate step.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
EDAA	CDZT	RSL	Convert from Zoned (Long←Zoned)	EDAB	CXZT	RSL	Convert from Zoned (Extended←Zoned)
EDA8	CZDT	RSL	Convert to Zoned (Zoned←Long)	EDA9	CZXT	RSL	Convert to Zoned (Zoned←Extended)

Table 356. Instructions converting between decimal floating-point and zoned decimal

All the instructions have the format shown in Table 357:

opcode	L_2	B_2	D_2	R_1	M_3	opcode

Table 357. Format of DFP/zoned decimal conversion instructions

The decimal floating-point operand is in FPR R_1, and the zoned decimal operand is in memory at the Effective Address of the second operand.

The machine instruction statement format of the instructions is

```
mnemonic  R1,D2(N2,B2),M3
```

where N_2 is the *true* number of bytes in the second operand. (Remember from Section 24.2 on page 366 that *you* specify the true length in the Assembler Language machine instruction statement, but the CPU uses the *encoded length* in the machine language instruction, which is one less.)

Because the decimal floating-point operand is at most 16 or 34 digits long, the CPU requires that the length of the zoned decimal operand satisfy

$0 \le L_2 \le 15$ (or $1 \le N_2 \le 16$) for Long DFP
$0 \le L_2 \le 33$ (or $1 \le N_2 \le 34$) for Extended DFP

The Convert from Zoned instructions are simpler: the N_2 zoned decimal digits at the second operand are converted to decimal floating-point; the Condition Code is unchanged.

The leftmost bit of the M_3 field determines the treatment of a possible sign code in the rightmost byte of the zoned operand:

- If M_3 = B'1000', the zoned operand is assumed to have a standard sign code (X'A,C,E,F' for +, (X'B,D' for −); the decimal floating-point operand is given that sign.
- If M_3 = B'0000', the zoned operand is assumed to have no sign code, so the sign bit of the decimal floating-point operand is set to zero to indicate plus. (This is important for numeric ASCII characters that have zone digit X'3' that would otherwise cause a decimal data exception.)

The preferred quantum of the result is one, meaning that the exponent assigned is set to zero so the result is treated as an integer.

Figure 454 shows the effect of the two values of the M_3 operand:

```
         CDZT  0,ZoneA,B'1000'    c(FPR0) = X'A2380000000049C5'
         CDZT  0,ZoneA,B'0000'    c(FPR0) = X'22380000000049C5'
         - - -
ZoneA    DC    Z'-12345'          X'F1F2F3F4D5'
```

Figure 454. Effect of the mask operand on Convert from Zoned results

The first instruction generates a negative result because M_3 = B'1000', while the second instruction ignores the sign code in the last zoned byte and returns a positive result.

The Convert to Zoned instructions convert the specified number of *rightmost* digits of the decimal floating-point operand to zoned, with the same restrictions on L_2 (and N_2).

The bits of the M_3 operand control how zone and sign digits are assigned; for convenience, the bits are designated SZPF from left to right.

S is bit zero of M_3: B'S...'. If S=0, the zoned result does not have a sign, and all zone digits are determined by the Z bit. If S=1, the zoned operand has a sign; its form is controlled by the P and F bits.

Z is bit one of M_3: B'.Z..'. If Z=0, the zone fields of the result are X'F'; if Z=1, the zone fields are X'3'. If S=1, the final zone (the sign) depends on the P and F bits.

P is bit two of M_3: B'..P.'. If S=0, the P bit is ignored (assumed to be zero) and the sign code depends on the Z bit. Otherwise, if P=0, the sign code for non-negative results is set to X'F'; if P=1, the plus sign code is X'C'.

F is bit three of M_3: B'...F'. If S=0, the F bit is ignored (assumed to be zero). Otherwise, if F=0, nothing happens. If F=1 *and* the absolute value of the zoned result is zero, the sign code is set to a plus code as determined by the P bit. (This is so that the zoned result can't be a negative zero.)

The field allocated for the zoned result may be too short for all the significant digits of the decimal floating-point operand. Because only the requested number of *rightmost* digits are converted, it's possible that significant digits could be lost. (This is one reason why the F bit tests for a zero result.)

The Convert to Zoned instructions set the Condition Code, as shown in Table 358:

CC	Meaning
0	Source operand is zero
1	Source operand is less than zero
2	Source operand is greater than zero
3	Source operand is infinity, QNaN, SNaN, or a partial result was generated

Table 358. Condition Code settings for Convert to Zoned

To illustrate, suppose we convert these DFP operands to zoned:

```
         LD    0,=DD'-12345'
         CZDT  0,ZoneB,B'0000'    Result = X'F0F1F2F3F4F5', CC=1
         CZDT  0,ZoneB,B'1000'    Result = X'F0F1F2F3F4D5', CC=1
         CZDT  0,ZoneB,B'0100'    Result = X'303132333435', CC=1
         CZDT  0,ZoneB,B'1100'    Result = X'3031323334D5', CC=1

         LD    0,=DD'+12345'
         CZDT  0,ZoneC,B'1010'    Result = X'F0F1F2F3F4F5', CC=2
         CZDT  0,ZoneC,B'0000'    Result = X'F0F1F2F3F4F5', CC=2
         CZDT  0,ZoneD,B'0000'    Result = X'F2F3F4F5',     CC=3

         LD    0,=DD'-12000000'
         CZDT  0,ZoneD,B'1000'    Result = X'F0F0F0D0',     CC=3
         CZDT  0,ZoneD,B'1001'    Result = X'F0F0F0C0',     CC=3
         - - -
ZoneB    DS    ZL6
ZoneC    DS    ZL6
ZoneD    DS    ZL4
```

Figure 455. Examples of converting decimal floating-point to zoned

Most of the results in Figure 455 are what you might expect, but some may look unusual:

- In the first group, the fourth CZDT instruction's M_3 digit is B'1100': the S bit means the result should be signed, and because the result is negative, the Z, P, and F bits don't apply.
- In the second group, the first CZDT instruction's M_3 digit is B'1010': the S bit means the result should be signed, the Z bit is zero so the result's zone is X'F' independent of the P and F bits. The third CZDT instruction sets the CC=3 because a partial result was generated.
- In the third group, both CZDT instruction set CC=3 because the result is partial.

Exercises

35.10.1.(2)+ If you are converting decimal floating-point data to signed packed decimal, what instructions can you use to test beforehand whether or not any high-order digits might be lost?

35.10.2.(2) Why do CUDTR and CUXTR need no M_4 operand?

35.10.3.(2)+ In Figure 449 on page 710, what is the LeftMost Digit (LMD) of the decimal floating-point result in (FPR0,FPR2)? Why?

35.10.4.(2) Can a source operand of CZDT be zero and cause Condition Code 3? If yes, give an example; if not, explain why.

35.10.5.(3)+ Make a table showing the effect of all possible combinations of the SZPF bits in the M_3 field of a CZDT instruction.

35.11. Decimal Floating-Point Load Operations

This section describes four groups of instructions; all the instructions are register-register operations.

- Load and Test, Load Complement, Load Negative, and Load Positive
- Load Floating-Point Integer
- Load Lengthened
- Load Rounded

35.11.1. Load and Test, Complement, Negative, and Positive

The instructions in Tables 359 and 361 starting on page 715 have operand format R_1,R_2.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3D6	LTDTR	RRE	Load and Test (Long)	B3DE	LTXTR	RRE	Load and Test (Extended)

Table 359. Decimal floating-point Load and Test instructions

If the source operand is a QNaN, the CC is set to 3; if it is a SNaN, an invalid operation exception occurs; if masked off, the corresponding QNaN is placed in the first-operand location.

Table 360 shows the Condition Code setting.

CC	Meaning
0	Operand is zero
1	Operand is less than zero
2	Operand is greater than zero
3	Operand is a NaN

Table 360. CC setting after DFP Load and Test instructions

The three instructions in Table 361 set the sign bit as indicated by the instruction name.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B373	LCDFR	RRE	Load Complement (Long)	B371	LNDFR	RRE	Load Negative (Long)
B370	LPDFR	RRE	Load Positive (Long)				

Table 361. Instructions copying/complementing data between FPRs

These instructions can be used for both decimal and binary floating-point operands: they are type-insensitive and don't care if the operand is a NaN or infinity. Unlike the corresponding instructions for hexadecimal floating-point operands, they *don't* change the Condition Code.

Some examples of these instructions, showing that only the sign bit changes:

```
LD    0,=DD'-21.43'     c(FPR0) = X'A2300000000008C3'
LCDFR 1,0               c(FPR1) = X'22300000000008C3'
LNDFR 2,1               c(FPR2) = X'A2300000000008C3'
LPDFR 3,2               c(FPR3) = X'22300000000008C3'
```

35.11.2. Load Floating-Point Integer

The two Load Floating-Point Integer instructions in Table 362 convert a decimal floating-point second operand to an integer-valued first operand in the same format. The CC is unchanged.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3D7	FIDTR	RRF	Load Floating-Point Integer (Long)	B3DF	FIXTR	RRF	Load Floating-Point Integer (Extended)

Table 362. Decimal floating-point Load Floating-point Integer instructions

The instructions have two mask fields, M_3 and M_4, shown in Table 363:

opcode	M_3	M_4	R_1	R_2

Table 363. Format of Load FP Integer instructions

The machine instruction statement format specifies the operands in this order:

```
mnemonic  R1,M3,R2,M4
```

The M_3 mask controls rounding of the delivered operand; its possible values are given in Table 342 on page 702.

The M_4 mask controls recognition of the inexact exception. If the value of the result is different from the value of the second operand it is inexact; then, if bit 1 (B'0100') of M_4 is one, no inexact exception will occur.

If the second operand is a QNaN, so is the result. If the second operand is a SNaN, an invalid operation exception occurs; if masked off, the corresponding QNaN is the result. To illustrate:

```
LD    6,=DD'7.2'        Fractional operand X'2234000000000072'
FIDTR 2,B'1000',6,4     No exception:      X'2238000000000007'

LD    6,=DD'7.2'        Same fractional operand
FIDTR 2,B'1000',6,0     Same result, inexact exception
```

35.11.3. Load Lengthened

Table 364 lists the two instructions that convert a decimal floating-point operand to a longer format. They can be used to convert from the short (storage) format to long format, or from long to extended format for computations. (Converting the short format to extended requires two instructions.)

Because a longer format has greater precision and range, no rounding is needed. Only infinity and SNaNs need special treatment; QNaNs are converted to the longer format with the payload (if any) extended with zeros on the left. Neither instruction changes the CC.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3D4	LDETR	RRF	Load Lengthened (Short to Long)	B3DC	LXDTR	RRF	Load Lengthened (Long to Extended)

Table 364. Decimal floating-point Load Lengthened instructions

They have the instruction format illustrated in Table 350 on page 708, and machine instruction statement format

```
mnemonic  R1,R2,M4
```

The second operand is extended to the next longer format, and placed in the first operand register.

Table 365 shows how the M_4 mask controls the treatment of an infinity or SNaN.

Operand	M_4	Action
Infinity	0	Canonical infinity for the target format
	8	Source infinity with payload padded with zeros on the left
SNaN	0	Invalid operation exception; if masked, deliver corresponding QNaN with payload padded with zeros on the left
	8	Source SNaN with payload padded with zeros on the left

Table 365. Load Lengthened special operand control mask

For finite values, the M_4 mask makes no difference:

```
         LE    2,=ED'(QNaN)'     Short source operand X'7C000000'
         LDETR 4,2,0             Long result  X'7C00000000000000'

         LE    2,=ED'123.4567'   Short source operand X'2614D2E7'
         LDETR 4,2,0             Long result  X'222800000014D2E7'
         LXDTR 4,4,0             Extended result in FPR4, FPR6 is
*        c(FPR4,FPR6) = X'2207000000000000 000000000014D2E7'

         LD    1,=DD'7654.321'   Long operand X'222C0000007D51A1'
         LXDTR 5,1,0             Extended result in FPR5, FPR7 is
*        c(FPR5,FPR7) = X'2207400000000000 00000000007D51A1'
```

Note that the second operand of LDETR has short format.

35.11.4. Load Rounded

The two Load Rounded instructions in Table 366 convert an operand to the next shorter format: extended to long, and long to short. Because the shorter format has less precision and narrower range, the M_3 rounding mask field is provided in the instruction, as shown in Table 342 on page 702. If the source operand is finite, these instructions can indicate underflow, overflow, and inexact exceptions.

Neither instruction changes the CC.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3D5	LEDTR	RRF	Load Rounded (Long to Short)	B3DD	LDXTR	RRF	Load Rounded (Extended to Long)

Table 366. Decimal floating-point rounding/lengthening instructions

If the source operand's range causes an overflow or underflow condition when rounded to the shorter format, an interruption will occur if the mask bits in the Floating-Point Control Register are one. If the appropriate mask bit is zero, the exponent is scaled to be in range.

The B'1000' bit in the M_4 mask controls the treatment of infinity and SNaNs:

- If the source operand is infinity and the M_4 bit is zero, the result is a canonical infinity; and if the M_4 bit is one, trailing exponent bits are set to zero and the payload is shortened by removing high-order digits.
- If the source operand is a SNaN and the M_4 bit is zero, an invalid operation condition occurs; if the interruption is masked off, the result is the corresponding QNaN with shortened payload. If the M_4 bit is one, the invalid operation condition is suppressed and the result is a canonicalized SNaN with shortened payload.

Note that LDETR and LEDTR are the only DFP instructions that operate on short DFP operands. (TDCET tests a short operand, but doesn't operate on it.) Arithmetic on short-format data requires lengthening the source operands and rounding the result back to short format. Suppose we need the short-format quotient of 24÷7:

```
LE    2,=ED'24'          Load first operand    X'22500024'
LDETR 2,2,B'0000'        Lengthen it   X'2238000000000024'
LE    6,=ED'7'           Load second operand   X'22500007'
LDETR 6,6,B'0000'        Lengthen it   X'2238000000000007'
DDTR  2,2,6              Long quotient X'2DFE28BC628BC629'
LEDTR 2,B'0000',2,0      Round to short format X'2DF8A2F1'
STE   2,Quotient         Store the short result
```

Figure 456. DFP arithmetic with short operands

Both the division and rounding instructions indicate an inexact result; 24÷7 is not a finite fraction.

Exercises

35.11.1.(3) Suppose the source operand of a Load Rounded instruction is a SNaN. Show the possible combinations of M_4 mask bit and FPCR invalid-operation mask, and the delivered result for each case.

35.11.2.(1) With reference to Figure 456, what *is* the true value of 24÷7?

35.11.3.(1)+ Table 361 on page 715 has no instructions for extended precision operands. How then could you do a “Load Complement” of an extended precision operand?

35.11.4.(2) Under what circumstances would you use a Test Data Class instruction in preference to a Load and Test instruction?

35.12. Decimal Floating-Point Miscellaneous Operations (*)

This subsection describes some operations of (possibly) less interest or less frequent use:

- Set Decimal Rounding Mode
- Extract and Insert Biased Exponent
- Extract Significance
- Shift Coefficient Left and Right
- Quantize
- Reround
- Test Data Group

35.12.1. Set Decimal Rounding Mode

The SRNMT instruction in Table 367 sets the DFP rounding mode.

Op	Mnem	Type	Instruction
B2B9	SRNMT	S	Set Decimal Rounding Mode

Table 367. Decimal floating-point Set Rounding Mode instruction

The machine instruction statement is written with a single operand:

SRNMT $D_2(B_2)$

The rightmost three bits of the Effective Address are placed in the FPCR's rightmost byte, in the bits indicated by **DRM** in Figure 457. The CC is unchanged.

Comparing this figure to Figure 413 on page 649, the only difference is the presence of the “q” mask and flag bits (for the quantum exception), and the three added **DRM** bits for the decimal rounding mode.

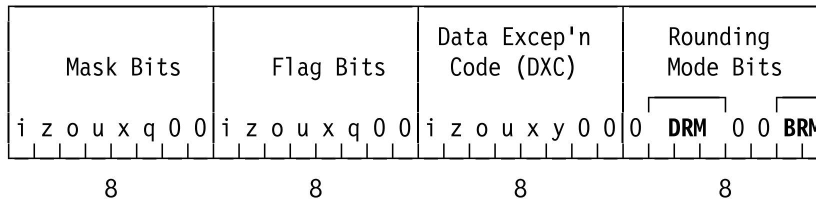


Figure 457. Floating-Point Control Register showing Decimal Rounding Mode bits

The rightmost 2 bits of the DRM field have the same meaning as for the Binary Rounding Mode (BRM):

BRM/DRM **Rounding Action**
B'000' To nearest (default; “ties” round to even),
B'001' Toward zero (truncate; “chop”)
B'010' Up (toward $+\infty$)
B'011' Down (toward $-\infty$)

There are four additional decimal floating-point rounding modes when the leftmost DRM bit is 1:

DRM **Rounding Action**
B'100' To nearest, with ties away from 0 (instead of “to even”)
B'101' To nearest, with ties toward 0
B'110' Round away from 0
B'111' Round to prepare for shorter precision

For example, to set the decimal rounding mode to the default, you could execute

```
SRNMT B'000'          Set DRM to unbiased round to nearest
```

35.12.2. Extract and Insert Biased Exponent

Because the representation of the true biased exponent of a DFP number is so complex, System z provides the instructions listed in Table 368 on page 719 to extract and to insert its value.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3E5	EEDTR	RRE	Extract Biased Exponent (Long)	B3ED	EEXTR	RRE	Extract Biased Exponent (Extended)
B3F6	IEDTR	RRF	Insert Biased Exponent (Long)	B3FE	IEXTR	RRF	Insert Biased Exponent (Extended)

Table 368. Decimal floating-point Insert/Extract Biased Exponent instructions

None of the instructions change the Condition Code.

Machine instruction statements for the extract instructions have two operands:

EEDTR R_1,R_2

where the R_2 floating-point register holds a decimal floating-point number and its biased exponent is placed in the 64-bit general register specified by R_1. For example:

```
         LD    5,=DD'2.5'          Long DFP operand X'2234000000000025'
         EEDTR 3,5                 Result in GG3 = X'18D' = 397
         AGHI  3,-398              Remove exponent bias
*                                  c(GG3) = -1, the unbiased exponent
```

Figure 458. Example of extracting DFP biased exponent

The value in FPR5 is 25×10^{-1}; the exponent bias for a long DFP number is 398, so the true exponent is $397-398 = -1$, as expected.

For special values, a negative value is placed in the R_1 general register, as shown in Table 369:

DFP operand	Binary result
Infinity	−1
QNaN	−2
SNaN	−3

Table 369. Extracted Biased Exponent for DFP special values

Thus, the EEDTR and EEXTR extract instructions can be applied to any DFP operand in a floating-point register.

The instructions that insert a biased exponent have three operands:

IEDTR R_1,R_3,R_2

where the source DFP operand is in floating-point register R_3, the new biased exponent is in 64-bit general register R_2, and the result is placed in floating-point register R_1. For example:

```
         LD    5,=DD'2.5'          Source operand = X'2234000000000025'
         LG    9,=FD'399'          New biased exponent
         IEDTR 2,5,9               Insert; result = X'223C000000000025'
```

Figure 459. Example of inserting a biased DFP exponent

In this case, the significand is unchanged, and the new biased exponent (399) means that the new true exponent is $399-398 = +1$, so the value of the number in FPR2 is $25\times10^{+1}$, or 250.

Thus, the Insert Biased Exponent instructions provide an easy way to multiply or divide finite numbers by a power of 10, without changing significance.

Special values are treated in a way much like that for the extract instructions. The results are illustrated in Table 370 on page 720, where “CMax” represents the largest characteristic (biased exponent) value.

R_2 operand	DFP source operand in R_3			
	Finite	**Infinity**	**QNaN**	**SNaN**
Value > CMax	QNaN	QNaN	QNaN	QNaN
0 ≤ Value ≤ CMax	Finite	Finite	Finite	Finite
Value = −1	Infinity	Infinity	Infinity	Infinity
Value = −2	QNaN	QNaN	QNaN	QNaN
Value = −3	SNaN	SNaN	SNaN	SNaN
Value ≤ −4	QNaN	QNaN	QNaN	QNaN

Table 370. DFP Insert Biased Exponent results

35.12.3. Extract Significance

The instructions in Table 371 let you determine the number of significant digits in a DFP operand. Neither instruction changes the Condition Code.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3E7	ESDTR	RRE	Extract Significance (Long)	B3EF	ESXTR	RRE	Extract Significance (Extended)

Table 371. Decimal floating-point Extract Significance instructions

The second operand of the machine instruction statement is the DFP number, and the result is placed in the 64-bit R_1 general register. For example:

```
LD    0,=DD'2.5'        2 digits, value = X'2234000000000025'
ESDTR 0,0               c(GG0) = 2
LD    1,=DD'2.5E10'     2 digits, value = X'225C000000000025'
ESDTR 1,1               c(GG1) = 2
LD    2,=DD'2.5000000'  8 digits, value = X'221C000002500000'
ESDTR 2,2               c(GG2) = 8
```

Figure 460. Examples of DFP Extract Significance instructions

Special values return zero (for zeros), −1 for infinities, −2 for QNaNs, and −3 for SNaNs.

35.12.4. Shift Significand Left/Right

Table 372 on page 721 lists the four instructions for shifting decimal floating-point operands left and right.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED40	SLDT	RXF	Shift Significand Left (Long)	ED48	SLXT	RXF	Shift Significand Left (Extended)
ED41	SRDT	RXF	Shift Significand Right (Long)	ED49	SRXT	RXF	Shift Significand Right (Extended)

Table 372. Decimal floating-point Shift Significand instructions

All four leave the CC unchanged, and cause no exceptions.

These shifts are similar to the logical shift instructions for binary data in the general registers: digits shifted off either end are lost, and zeros are inserted into positions vacated by the shift. Exponents are unchanged, and the results are not rounded.

The instruction format[235] is illustrated in Table 373 on page 721:

opcode	R_3	X_2	B_2	D_2	R_1		opcode

Table 373. Format of DFP shift instructions

The operands of the assembler instruction statements have format R_1,R_3,D_2(X_2,B_2), where the source operand is in FPR R_3, the result is placed in FPR R_1, and the shift amount is determined by the rightmost 6 bits of the Effective Address. For example, shifting right by one digit effectively divides the source operand by 10, truncating the lost digits:

```
         LD    0,=DD'1000000'     Source = X'2238000000100000'
         SRDT  0,0,1              Result = X'2238000000020000'

         LD    0,=DD'1234567'     Source = X'223800000014D2E7'
         SRDT  0,0,1              Result = X'2238000000028E56'
```

Left shifts are similar, except that the high-order digit of the significand moves into the Combination Field; the previous occupant is lost. For example:

```
         LD    0,=DD'1000000000'  Source = X'2238000040000000'
         SLDT  0,0,6              Result = X'2638000000000000'
*                                 Leftmost digit now in Combo Field

         LD    0,=DD'1000000000'  Source = X'2238000040000000'
         SLDT  0,0,7              Result = X'2238000000000000'
*                                 Zero significand, all digits lost
```

In the first case, the significant digit is shifted into the Combination Field, and in the second case it is shifted completely out leaving a zero significand with exponent +9.

Because the exponent is not changed, you might consider these instructions as a way to multiply or divide by 10; but be cautious. The *significance* of the operand is changed depending on the direction and amount of the shift.

The significands of infinity, SNaN, and QNaN are shifted without regard to the type of operand, and without causing any exceptions.

With these shift instructions, we can now convert all 34 digits of an extended decimal floating-point value to packed decimal, using instructions like those in Figure 461 on page 722.

```
         LD    0,=LD'1234567890123456789012345678901234'   34 digits
         LD    2,=LD'1234567890123456789012345678901234'+8
         CSXTR 4,0,0              Convert 31 signed digits to (GG4,GG5)
         STMG  4,5,Packed+2       Store low-order 31 digits and sign
         SRXT  0,0,31             Shift, keeping 3 high-order digits
         CUXTR 4,0                Convert 3 unsigned digits
         STH   5,Packed           Store leading 0, remaining 3 digits
         - - -
Packed   DS    XL18               34 Packed digits and sign
```

Figure 461. Converting an extended decimal floating-point value to packed decimal

Note that CSXTR requires that the R_1 operand specify an even-odd pair of 64-bit registers.

The result at **Packed** is X'01234567890123456789012345678901234C'. Because the symbol **Packed** may not be aligned on a halfword preceding a doubleword boundary, the Assembler may diagnose a possibly unfavorable operand alignment; such messages can be avoided by preceding the DS instruction with

235 This format is unusual in placing the R_3 operand in the position where most instructions put the R_1 operand.

```
         ORG   *+2,8,-2            Halfword preceding doubleword
```

You can consult the *High Level Assembler Language Reference* for details.

35.12.5. Quantize

The quantize instructions in Table 374 do a very simple operation: they make one operand have the same decimal point alignment as another.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F5	QADTR	RRF	Quantize (Long)	B3FD	QAXTR	RRF	Quantize (Extended)

Table 374. Decimal floating-point Quantize instructions

The instruction format in Table 375 has four operands:

opcode	R_3	M_4	R_1	R_2

Table 375. Format of decimal floating-point Quantize instructions

and the assembler instruction statement is written

```
         QADTR R1,R3,R2,M4
```

The source operand is in FPR R_3 and the result is placed in FPR R_1; the operand in FPR R_2 (the "quantizing" operand) provides the quantum to be assigned to the R_3 operand. The M_4 rounding mask controls the rounding of the result, and takes the values listed in Table 342 on page 702. The Condition Code is unchanged.

The idea is to make the result have the same exponent (and quantum) as the R_2 operand, preserving the value of the R_3 operand to the greatest possible extent.

For example, suppose we have two values, 12789 (12789×10^0) and 12.789 (12789×10^{-3}).

```
A        DC    DD'12789'           Generates X'2238000000004BCF'
B        DC    DD'12789E-3'        Generates X'222C000000004BCF'
```

Both have the same significand, 12789, but their exponents differ by 3. Now, suppose we want to quantize A to have the same quantum/exponent as B, while preserving the value of A.

```
         LD    0,A                c(FPR0) = X'2238000000004BCF'
         LD    2,B                c(FPR2) = X'222C000000004BCF'
         QADTR 4,0,2,B'1000'      c(FPR4) = X'222C0000012F3C00'=12789000
```

The result in FPR4 has the same quantum/exponent as B, but the significand 12789000 has effectively been shifted left three digits and is now larger by 1000. Because the result has B's exponent (-3), its value is $12789000.\times10^{-3}$, the same as the value of A.

Now, consider quantizing B to have the same quantum/exponent as A.

```
         LD    0,A                c(FPR0) = X'2238000000004BCF'
         LD    2,B                c(FPR2) = X'222C000000004BCF'
         QADTR 4,2,0,B'1000'      c(FPR4) = X'2238000000000013'=13
```

Because the result in FPR4 must have the same quantum as A, the significand of B must effectively be shifted right by three digits. The rounding mask causes 12.789 to be rounded to 13×10^0.

Infinity as either the source or quantizing operand produces an invalid operation exception; if masked off the result is a canonical QNaN:

```
         LD    0,A                 Finite value in FPR0
         LD    2,=DD'(Inf)'        Infinity in FPR2
         QADTR 4,0,2,B'0000'       Quantize A with infinity; c(FPR4) = QNaN
         QADTR 6,2,0,B'0000'       Quantize infinity with A; c(FPR6) = QNaN
```

Quantizing with QNaNs always produces QNaNs. Quantizing with SNaNs produces an invalid operation exception; if masked off, the result is the corresponding QNaN.

These examples illustrate the rules for quantizing operations:

1. If the exponent of the result is being decreased (as in the first example), *and* the result does not have more digits than supported by the operand format, the result has the same value as the source operand. But if the result would have *more* digits than supported, an invalid operation is signaled; if masked off, the result is a default QNaN.

   For example:

   ```
   LD    0,A                 Source operand = X'2238000000004BCF'=12789
   LD    2,=DD'12789E-12'    Second operand = X'2208000000004BCF'
   QADTR 4,0,2,B'1000'       Invalid operation
   ```

   If the quantum/exponent of the result in FPR4 were to be the same as the second operand's, the significand of A would have to be shifted left 12 digits, giving 12789000000000000. But this is 17 digits long, too many digits for a long decimal floating-point operand.

2. If the exponent of the result is being increased (as in the second example), the result is rounded according to the M_4 rounding modifier. This can possibly generate an inexact exception if the value of the result is different from the value of the source operand. A zero value with nonzero exponent can be generated.

3. If the source or quantizing operand is infinity, the result is a canonical infinity.

Because the exponent of the result is the same as that of the second (quantizing) operand, no overflow or underflow occurs. A finite result always has the requested quantum/exponent.

To summarize the exception conditions for finite operands:

- If the requested quantum/exponent (from the quantizing operand) is smaller than the quantum of the source operand, the only possible exception is an invalid operation exception that might occur if the significand would have too many digits.
- If the requested quantum/exponent (from the quantizing operand) is greater than the quantum of the source operand, the only possible exception is an inexact exception that might occur if nonzero digits were lost.

Here's a practical example using a quantize instruction. Suppose you must calculate the final price of an item costing $923.85 after adding tax of 8.25 percent. You would probably start with something like this:

```
         LD    0,Price           Get item price (923.85)
         LD    2,Tax             Load tax rate (1.0825)
         MDTR  0,2,0             Calculate price+tax (1000.067625)
         STD   0,Cost            Store final cost (1000.067625)
         - - -
Price    DC    DD'923.85'        Item price
Tax      DC    DD'1.0825'        Tax rate + 1 for price
Cost     DS    DD                Calculated item cost
```

Figure 462. Calculate price plus tax

The product was calculated by the MDTR instruction to 16 digits, and rounded according to the current rounding mode. But you will want to display the cost with only two decimal places, as $1000.07; because it's important to avoid double rounding, how should you proceed?

First, calculate the product without rounding, by setting the Decimal Rounding Mode to “prepare for shorter precision”:

```
         SRNMT 7                 Set rounding mode to B'111'
```

This rounding mode selects the result of the MDTR instruction with the smaller magnitude, unless the rounding digit is 0 or 5, when the result with the largest magnitude is chosen.

The calculated result has too many digits, so we must round the result to two decimal digits. But since we don't know how many digits are to the left of the decimal point, we can use a quantize operation. We know that the original price had two decimal digits, so we can use it as the quantizing operand:

```
SRNMT 7                  Set rounding mode to B'111'
LD    4,Price            Item price (923.85) in FPR4
LD    2,Tax              Tax rate (1.0825) in FPR2
MDTR  2,2,4              Price+tax (1000.067625) in FPR2
QADTR 0,2,4,B'1100'      Quantize to 2 decimal digits
STD   0,Cost             Store final cost (1000.07)
- - -                    Convert and format for display
```

Figure 463. Correctly rounding a cost to two decimal digits

The rounding mask (12) of the QADTR instruction rounds to nearest with ties away from zero; this is the familiar decimal rounding used for many business calculations.

35.12.6. Reround

The instructions in Table 376 perform a reround operation. In effect, rerounding simply changes the number of significant digits.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
B3F7	RRDTR	RRF	Reround (Long)	B3FF	RRXTR	RRF	Reround (Extended)

Table 376. Decimal floating-point Reround instructions

The instruction format is shown in Table 375 on page 722, and assembler instruction statements are written as shown following that table. Neither instruction changes the Condition Code.

These instructions let you simulate a DFP operation with a different precision from any supported format, and with the effect of a single rounding. You start by specifying “Round to prepare for shorter precision” in the operation, and then you follow that with the desired rounding method in the reround instruction.

The reround instructions work like this:

- The desired significance is an unsigned integer in GPR R_2.
- The operand in FPR R_3 is rounded to the significance specified in GPR R_2, and the result is placed in FPR R_1.
- If the requested significance is less than or equal to the significance of the third operand, it is placed unchanged in R_1.
- If the requested significance is greater than the significance of the third operand, it is rounded to the requested significance using the rounding mask specified by the M_4 operand.
- If the instruction can't represent the result to the requested significance, or if the source operand is a SNaN or infinity, an invalid operation is signaled. If the exception is disabled, the delivered result is a default QNaN.
- As with other operations, if the result differs in value from the source operand in R_3, an inexact exception is signaled.

These instructions may seem strange: you already have rounding options for all the arithmetic operations, so why would you want to use a “reround” instruction?

Suppose you are offering to do some work, and you calculate your total cost as $1783. You probably prefer to quote a slightly larger “rounder” number, such as $1800. You can do this with a reround instruction:

```
         LD    4,=DD'1783'          Calculated total cost
         LA    5,2                  2 significant digits
         RRDTR 6,4,5,B'1110'        Round to 2 digits, away from zero
         SLDT  6,6,2                Shift left 2 digits
         - - -                      Convert result to printable form
```

Figure 464. Example of a reround instruction

and the result in FPR6 is 1800, the value to be quoted.

This example works for a total cost with at most 4 digits, but you would like to use it for larger values. Supposing that you still want to quote values with two low-order zero digits, you can modify the example in Figure 464:

```
Final0s  Equ   2
         LD    4,=DD'1783333'       Calculated total cost = $1,783,333
         ESDTR 5,4                  Extract its significance into GG5
         AHI   5,-Final0s           Reduce by number of final zeros
         RRDTR 6,4,5,B'1110'        Round to to specified significance
         SLDT  0,6,Final0s          Compensate for digits removed
         - - -                      Convert to printable format
```

Figure 465. Example of rerounding arbitrary amounts

Because we have rounded away from zero, the result in FPR0 is 1783400. (For large values like this you would probably want to set **Final0s** to 3, to round to thousands.)

Because the reround instructions can round longer operands to shorter, the Assembler supports a "round for reround" modifier. These 17-digit constants have an even final digit and rounding digits 4, 5, and 6. You can see that the R15 rounding option leaves the final digit unchanged.

```
         DC    DD'12345678901234564R8'  Generates X'263D34B9C1E28E56'
         DC    DD'12345678901234565R8'  Generates X'263D34B9C1E28E56'
         DC    DD'12345678901234566R8'  Generates X'263D34B9C1E28E57'

         DC    DD'12345678901234564R15' Generates X'263D34B9C1E28E56'
         DC    DD'12345678901234565R15' Generates X'263D34B9C1E28E56'
         DC    DD'12345678901234566R15' Generates X'263D34B9C1E28E56'

         DC    DD'12345678901234574R8'  Generates X'263D34B9C1E28E57'
         DC    DD'12345678901234575R8'  Generates X'263D34B9C1E28E58'
         DC    DD'12345678901234576R8'  Generates X'263D34B9C1E28E58'

         DC    DD'12345678901234574R15' Generates X'263D34B9C1E28E57'
         DC    DD'12345678901234575R15' Generates X'263D34B9C1E28E57'
         DC    DD'12345678901234576R15' Generates X'263D34B9C1E28E57'
```

Figure 466. Examples of assembled DFP constants using rounding for reround

These constants are now prepared for rerounding if desired.[236]

35.12.7. Decimal Floating-Point Data Groups (*)

You may find in evaluating a complex expression that you need to do parts of the calculation in a longer format, and want to know whether the result will have the same value and quantum as if the calculation had been done in the shorter format. The Test Data Group instructions in Table 377 on page 726 provide this information.

236 Most applications would probably define separate constants of the required length, rather than rounding longer ones.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
ED51	TDGET	RXE	Test Data Group (Short)	ED55	TDGDT	RXE	Test Data Group (Long)
ED59	TDGXT	RXE	Test Data Group (Extended)				

Table 377. Decimal floating-point Test Data Group instructions

Like the Test Data Class instructions in Section 35.4 on page 693, you can test the properties of an operand without causing any exceptions. The assembler instruction statement is written

```
mnemonic R1,D2(X2,B2)
```

The rightmost 12 bits of the second operand's Effective Address test the operand in R_1. Using the same bit numbering as in Table 329 on page 694, the tested groups are shown in Table 378.

+ sign	– sign	Group
52	53	Zero with non-extreme exponent
54	55	Zero with extreme exponent
56	57	Normal or subnormal with extreme exponent
58	59	Normal with non-extreme exponent and zero leftmost digit
60	61	Normal with non-extreme exponent and nonzero leftmost digit
62	63	Infinity or NaN

Table 378. Test Data Group second-operand bits

If a 1-bit in the Effective Address matches one of the properties in the table, the Condition Code is set to 1; otherwise, it is set to 0.

The main use of these instructions is checking an operand to verify it has these properties:

- its value is finite,
- its exponent is not extreme (neither maximum nor minimum),
- its leftmost significant digit is zero,
- calculating the operand in a wider format would have produced the same value and quantum.

If the result satisfies these conditions, it is said to be "safe".

For example, suppose you multiply the number 100000000×10^5 by itself in long and extended arithmetic. Because it has 9 significant digits, the extended precision product has 17 significant digits:

```
LD    0,=LD'100000000E5'      c(FPR0)=X'2209400000000000'
LD    2,=LD'100000000E5'+8    c(FPR2)=X'0000000008000000'
MXTR  0,0,0                   Form extended product
```

The result in (FPR0,FPR2) is X'220A800000000000 0040000000000000' and the exponent is 10. When we form the same product in long precision:

```
LD    0,=DD'100000000E5'      c(FPR0)=X'224C000008000000'
MDTR  0,0,0                   c(FPR0)=X'2664000000000000'
```

where the most significant digit is also the leftmost digit, and the exponent is 11. If we test the data group of this result:

```
TDGDT 0,B'000000001100'       Test normal value, nonzero LMD
```

the Condition Code is set to 1, indicating that this would not be a "safe" result. The value of the result is the same, but the exponents (and therefore quanta) were different.

Exercises

35.12.1.(2) Assuming you start with the same operand in FPR2 for each shift instruction, what will be the result of extracting the significance of the result after executing each of these instructions?

```
         LD    2,=DD'12340000E3'  Initial value for each shift
         - - -
(1)      SLDT  2,2,0
(2)      SLDT  2,2,3
(3)      SRDT  2,2,6
(4)      SLDT  2,2,10
(5)      SRDT  2,2,8
```

35.12.2.(2) In Figure 460 on page 720, it appears that the third DFP operand (X'221C000002500000') has only seven significant digits, but the result of extracting its significance is 8. Why are they different?

35.12.3.(2) The shift instructions effectively multiply or divide the source operand by a power of 10. Why or why not use them in place of the "regular" multiply and divide instructions?

35.12.4.(2) Give an example showing how a quantize operation can generate a non-canonical zero, a zero result with nonzero exponent.

35.12.5.(1)+ In Figure 461 on page 722, why was the symbol **Packed** not defined as PL18?

35.13. Example of a Decimal Floating-Point "Business" Computation

We'll repeat the simple business computation that used packed decimal arithmetic in Section 29.11 on page 526, now using decimal floating-point arithmetic.

This is the situation: we suppose a wholesaler and a retailer complete an order.

1. The retailer orders from the wholesaler 60 high-tech widgets at $74.65 each.
   (60×$74.65 = $4479.00)
2. For this large order, the wholesaler discounts the price by 4.7%.
   ($4479.00×0.047 = $210.513, so the discounted price is $4479.00−$210.51 = $4268.49)
3. The wholesaler adds 9.75% local sales tax, and a $4.00 per-item shipping charge.
   (4268.49×1.0975 = 4684.67; the shipping charge is 60×$4.00 = $240.00, so the total is $4684.67+$240.00 = $4924.67.)
4. The retailer's pre-payment of $1000.00 is deducted. The result is the wholesaler's bill to the retailer.
   ($4294.67−$1000.00 = $3294.67)

Then, the retailer calculates his necessary markup and the sale price:

5. The retailer calculates his base cost for each item.
   ($4294.67/60 = $82.07785 or $82.08)
6. He then applies his retail markup (about 37%), adjusted to a sale cost just below one dollar.
   (The markup is $82.08×1.37 = $112.4496 or $112.45, and the adjustment is $0.54+$112.45 = $112.99)
7. Each item a customer buys must include 9.25% sales tax and a $7.50 recycling fee.
   (The sales tax is $112.99×1.0925 = $123.44; adding the recycling fee gives the final customer cost: $123.44+$7.50 = $130.94)
8. The result is the final cost per item to the customer.
9. The retailer's gross profit per item is (sale cost)−(base cost).
   ($112.99−$82.08 = $30.91. The percent gross profit is $30.91/$82.08 = 0.376 or 38%)

35.13.1. The Wholesaler's Calculation

We will use long decimal floating-point values. Because all arithmetic is done in the floating-point registers, we won't define constants with names, but use literals with the same values as in Section 29.11.1.

The values in the comments fields show the value of each operand as a decimal floating-point number.

```
LD     F1,=DD'60'          Number of items                           60.E+0
LD     F2,=DD'74.65'       Wholesale price per widget              7465.E-2
MDTR   F3,F1,F2            Price of all 60 items                 447900.E-2
LD     F4,=DD'.047'        Discount % for a large order              47.E-3
MDTR   F5,F3,F4            Discount amount                     21051300.E-5
SDTR   F3,F3,F5            Subtract the discount              426848700.E-5
LD     F6,=DD'1.0975'      Sales tax multiplier                   10975.E-4
MDTR   F3,F3,F6            Price with sales tax           4684664482500.E-9
LD     F7,=DD'4.00'        Shipping charge per item                 400.E-2
MDTR   F7,F7,F1            Shipping charge for 60 items           24000.E-2
ADTR   F3,F7,F3            Add shipping charge            4924664482500.E-9
LD     F8,=DD'1000.00'     Retailer's prepayment                 100000.E-2
SDTR   F3,F3,F8            Result = bill to retailer      3924664482500.E-9
```

So, we calculate that the final bill to the retailer is $3924.66. But this is one cent less than the packed decimal value! What happened?

Aha! We didn't use something like the "Shift and Round Decimal" (SRP) instruction to round intermediate results. So, we'll try again, using QADTR "quantize" instructions to round intermediate results to two decimal places, as was done with the SRP instructions in the packed decimal example.

```
LD     F1,=DD'60'          Number of items                           60.E+0
LD     F2,=DD'74.65'       Wholesale price per widget              7465.E-2
MDTR   F3,F1,F2            Price of all 60 items                 447900.E-2
LD     F4,=DD'.047'        Discount % for a large order              47.E-3
MDTR   F5,F3,F4            Discount amount                     21051300.E-5
QADTR  F5,F5,F2,B'0001' *  Round to two decimals                  21051.E-2
SDTR   F3,F3,F5            Subtract the discount                 426849.E-2
LD     F6,=DD'1.0975'      Sales tax multiplier                   10975.E-4
MDTR   F3,F3,F6            Price with sales tax              4684667775.E-6
QADTR  F3,F3,F2,B'0001' *  Round to two decimals                 468467.E-2
LD     F7,=DD'4.00'        Shipping charge per item                 400.E-2
MDTR   F7,F7,F1            Shipping charge for 60 items           24000.E-2
ADTR   F3,F7,F3            Add shipping charge                   492467.E-2
LD     F8,=DD'1000.00'     Retailer's prepayment                 100000.E-2
SDTR   F3,F3,F8            Result = bill to retailer             392467.E-2
```

Now, the calculated final bill to the retailer is $3924.67, matching the packed decimal calculation.

35.13.2. The Retailer's Calculation

Again, we'll use quantization instructions to mimic SRP rounding:

```
LD    F1,=DD'60'          Number of items                          60.E+0
LD    F2,=DD'4924.67'     Wholesale charge                      492467.E-2
LD    F3,=DD'1.37'        Retail markup                            137.E-2
LD    F4,=DD'.54'         Retail fudge factor                       54.E-2
LD    F5,=DD'.0925'       Retailer's sales tax                     925.E-4
LD    F6,=DD'7.50'        Recycle charge per item                  750.E-2
DDTR  F7,F2,F1            Base cost per item           8207783333333333.E-14
QADTR F7,F7,F3,B'0001'  * Round to 2 decimals                     8208.E-2
MDTR  F8,F7,F2            Calculate retail markup              1124496.E-4
QADTR F8,F8,F3,B'0001'  * Round to 2 decimals                    11245.E-2
ADTR  F8,F8,F4            Add retail fudge                       11299.E-2
MDTR  F9,F8,F5            Calculate sales tax                 10451575.E-6
QADTR F9,F9,F3,B'0001'  * Round to 2 decimals                     1045.E-2
ADTR  F10,F9,F8           Cost/item with sales tax               12344.E-2
ADTR  F10,F10,F6          Add recycle fee for cust cost          13094.E-2
SDTR  F8,F8,F7            Gross profit/item                       3092.E-2
DDTR  F8,F8,F7            Gross profit %               3765838206627680.E-16
QADTR F8,F8,F5,B'0001'  * Round % to xx.xx                        3766.E-4
```

We get the same results as we did with packed decimal: $130.94 for the total cost to the consumer, and a 37.66% profit margin for the retailer.

35.13.3. Comparing Packed and Floating Decimal

In the packed decimal calculations we had to keep track of the decimal point for each calculation. Furthermore, all packed decimal operations are storage-to-storage; on modern CPUs, the cost of memory references grows much faster than processor speed, so storage references are relatively expensive. Temporary work areas (as in Figures 323 and Figure 325 on page 528) add to the amount of storage needed, and intermediate results must often be copied to temporaries for later use.

Although neither the packed decimal nor the decimal floating-point example showed how the results would be formatted for printing, packed decimal formatting must “know” where to put the decimal point, and how many significant digits are in the value.

Decimal floating-point takes care of all the hard work for you! For example:

- All 16 floating-point registers are available, so temporary work areas in storage are rarely needed.
- Non-destructive operations minimize the need for copying data.
- Decimal floating-point knows where the decimal point is, and how many significant digits are present; instructions are available to easily determine both.
- A simple timing test showed that the decimal floating-point computation was several times faster than the packed decimal computation.

Comparing the two examples, you can see how programming decimal-arithmetic financial and business calculations can be much easier using decimal floating-point values and arithmetic.

35.14. Decimal Floating-Point Binary-Significand Format (*)

For other architectures

This format is not supported by System z.

We mentioned briefly on page 682 that a binary-integer significand could be used for decimal floating-point data. The IEEE 754-2008 standard defines this representation; it is intended mainly for implementations not using the complex hardware logic of the DPD formats.

The binary-significand encoding uses a simpler significand, and like the DPD representation, the Combination Field also mixes part of the characteristic and the leading significand digits.

As in the DPD representation, the first five bits of the CF are used to indicate infinity and NaNs:

- If the bits are 11110, the value is an infinity. In a canonical infinity, all remaining bits are zero.
- If the bits are 11111, the data item is a NaN; the following bit is zero for a QNaN and 1 for a SNaN. In a canonical NaN, all bits after the sixth are zero.

The standard defines the Combination Field to be w+5 bits long, where the five bits are reserved. This means that for short, long, and extended representations, w is 6, 8, and 11 respectively. The CF bits are denoted g_n, where n takes values from 0 to (5+w−1). If we call the significand T, then the value of a number in this format is defined this way:

1. If g_0 and g_1 are one of 00, 01, or 10, then the biased exponent E is formed from g_0 through $g_{(w+1)}$ and the significand is formed from bits $g_{(w+2)}$ through the end of the encoding (including T).
2. if g_0 and g_1 together are 11 and g_2 and g_3 are one of 00, 01, or 10, then the biased exponent E is formed from g_2 through $g_{(w+3)}$ and the significand is formed by prefixing the 4 bits $(8+g_{(w+4)})$ to T.
3. Exponent biases are the same as for the decimal encodings.

Figure 467 sketches this representation:

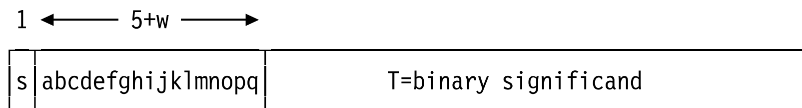


Figure 467. Example of DFP binary-significand format

Figure 468 shows a short-format value in this representation, where w=6:

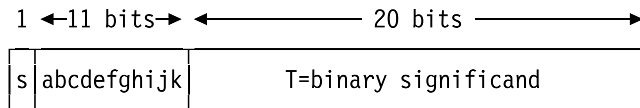


Figure 468. Sketch of short binary-significand format

Thus, if bits 'ab' are 00, 01, or 10, then the characteristic is bits 'abcdefgh', and the binary significand is formed from bits 'ijk' concatenated at the front of T (23 bits).

Similarly, if bits 'ab' are 11 and bits 'cd' are 00, 01, or 10, then the characteristic is bits 'cdefghij' and the binary significand is formed from the four bits (8+'k') concatenated at the front of T (24 bits).

We can derive the value of the short binary-encoded number X'3FFFFFFF' this way:

- The sign bit is 0, so the number is positive.
- The eleven bits of the Combination Field are B'011 1111 1111'. The 'ab' bits are 01, so the biased exponent is B'011 1111 1' or X'7F' = 127 decimal. Because the exponent bias is 101, the true exponent is 127−101 = 26.
- The significand is the remaining 3 bits of the CF, B'111', followed by the 20 bits of the TSF, giving X'7FFFFF', or 8388607 in decimal.

Thus, the value is $+8388607\times10^{26}$.

If J is the number of 10-bit, 3-BCD-digit declets in the Trailing Significand Field of a DPD-encoded number (2, 5, or 11), then the maximum valid value of the binary-encoded significand is the same as that of the corresponding decimal-encoded significand: $10^{(3\times J+1)}-1$ for numeric values, or $10^{(3\times J)}-1$ when T is used as the payload of a NaN. If the value exceeds this maximum, the significand is non-canonical and the value used is zero.

Computational operations in both decimal- and binary-significand representations produce only canonical significands, and always accept non-canonical significands as input operands.

Exercises

35.14.1.(3) Create a 32-bit binary-significand representation whose value exceeds the maximum value $10^{97}-1$.

35.14.2.(4) Construct a short-format binary-significand decimal floating-point number `Max`, the maximum valid value. Show its hexadecimal representation.

35.14.3.(2) Construct a short-format binary-significand decimal floating-point number `Min`, the minimum valid value. Show its hexadecimal representation.

35.14.4.(3) A binary-significand decimal floating-point number has representation `X'5FFFFFFF'`. Is it valid? If so, what is its value?

35.15. Summary

This section has covered a wide range of topics, some of which most programmers won't need to worry about. The key operations are the arithmetic, loading, and type-conversion instructions; the others are useful to know about, but aren't as frequently used.

We might describe decimal floating-point arithmetic this way:

Decimal floating-point arithmetic

It can behave like fixed-point arithmetic until it can't, and then it behaves like floating-point arithmetic.

Table 379 on page 732 lists the instructions that test data classes and data groups.

Function	Operand Length		
	Short	Long	Extended
Test Data Class	`TDCET`	`TDCDT`	`TDCXT`
Test Data Group	`TDGET`	`TDGDT`	`TDGXT`

Table 379. DFP Test Data Class and Test Data Group instructions

Table 380 on page 732 lists the decimal floating-point arithmetic and related instructions.

Function	Operand Length	
	Long	Extended
Add	ADTR	AXTR
Subtract	SDTR	SXTR
Divide	DDTR	DXTR
Multiply	MDTR	MXTR
Compare	CDTR	CXTR
Compare and Signal	KDTR	KXTR
Compare Biased Exponent	CEDTR	CEXTR
Load FP Integer	FIDTR	FIXTR
Extract Biased Exponent	EEDTR	EEXTR
Insert Biased Exponent	IEDTR	IEXTR
Extract Significance	ESDTR	ESXTR
Shift Significand Left	SLDT	SLXT
Shift Significand Right	SRDT	SRXT
Quantize	QADTR	QAXTR
Reround	RRDTR	RRXTR

Table 380. DFP Arithmetic and related instructions

Table 381 summarizes instructions that convert decimal floating-point data to and from different lengths and different data types.

To→ From↓	Short DFP	Long DFP	Ext. DFP	64-bit binary	Signed BCD	Unsigned BCD	Zoned decimal
Short DFP	—	LDETR	—	—	—	—	—
Long DFP	LEDTR	—	LXDTR	CGDTR	CSDTR	CUDTR	CZDT
Ext. DFP	—	LDXTR	—	CGXTR	CSXTR	CUXTR	CZXT
64-bit binary	—	CDGTR	CXGTR	—	—	—	—
Signed BCD	—	CDSTR	CXSTR	—	—	—	—
Unsigned BCD	—	CDUTR	CXUTR	—	—	—	—
Zoned decimal	—	CDZT	CZXT	—	—	—	—

Table 381. DFP length and type conversion instructions

Table 382 lists the decimal floating-point rounding and lengthening instructions.

Function	Round	Lengthen
Long to Short	LEDTR	
Short to Long		LDETR
Extended to Long	LDXTR	
Long to Extended		LXDTR

Table 382. DFP rounding and lengthening instructions

Table 383 on page 733 lists instructions that move data among the floating-point registers.

Function	Long	Extended
Copy Sign	CPSDR	
Load Complement	LCDFR	
Load Negative	LNDFR	
Load Positive	LPDFR	
Load and Test	LTDTR	LTXTR

Table 383. DFP data-loading instructions

Table 384 lists instructions that move data among the floating-point registers.

Function	Instruction
Copy FPR to GPR	LGDR
Copy GPR to FPR	LDGR

Table 384. Instructions copying between FPRs and GPRs

Table 385 lists the instruction that sets the decimal rounding model.

Function	Instruction
Set Decimal Rounding Mode	SRNMT

Table 385. Instruction setting decimal rounding mode

Table 386 summarizes the non-preferred declet encodings. (They are also shown in parentheses with all the declet encodings in Figure 470 on page 735.)

Noncanonical declets			BCD digits	Canonical declet
16E	26E	36E	888	06E
16F	26F	36F	889	06F
17E	27E	37E	898	07E
17F	27F	37F	899	07F
1EE	2EE	3EE	988	0EE
1EF	2EF	3EF	989	0EF
1FE	2FE	3FE	988	0FE
1FF	2FF	3FF	999	0FF

Table 386. Non-canonical declets

The following table shows how sets of three BCD digits are encoded into a canonical declet. The first two BCD digits are selected from a row, and the final BCD digit is selected from a column.

	0	1	2	3	4	5	6	7	8	9			0	1	2	3	4	5	6	7	8	9
00_	000	001	002	003	004	005	006	007	008	009		50_	280	281	282	283	284	285	286	287	288	289
01_	010	011	012	013	014	015	016	017	018	019		51_	290	291	292	293	294	295	296	297	298	299
02_	020	021	022	023	024	025	026	027	028	029		52_	2A0	2A1	2A2	2A3	2A4	2A5	2A6	2A7	2A8	2A9
03_	030	031	032	033	034	035	036	037	038	039		53_	2B0	2B1	2B2	2B3	2B4	2B5	2B6	2B7	2B8	2B9
04_	040	041	042	043	044	045	046	047	048	049		54_	2C0	2C1	2C2	2C3	2C4	2C5	2C6	2C7	2C8	2C9
05_	050	051	052	053	054	055	056	057	058	059		55_	2D0	2D1	2D2	2D3	2D4	2D5	2D6	2D7	2D8	2D9
06_	060	061	062	063	064	065	066	067	068	069		56_	2E0	2E1	2E2	2E3	2E4	2E5	2E6	2E7	2E8	2E9
07_	070	071	072	073	074	075	076	077	078	079		57_	2F0	2F1	2F2	2F3	2F4	2F5	2F6	2F7	2F8	2F9
08_	00A	00B	02A	02B	04A	04B	06A	06B	04E	04F		58_	28A	28B	2AA	2AB	2CA	2CB	2EA	2EB	2CE	2CF
09_	01A	01B	03A	03B	05A	05B	07A	07B	05E	05F		59_	29A	29B	2BA	2BB	2DA	2DB	2FA	2FB	2DE	2DF
10_	080	081	082	083	084	085	086	087	088	089		60_	300	301	302	303	304	305	306	307	308	309
11_	090	091	092	093	094	095	096	097	098	099		61_	310	311	312	313	314	315	316	317	318	319
12_	0A0	0A1	0A2	0A3	0A4	0A5	0A6	0A7	0A8	0A9		62_	320	321	322	323	324	325	326	327	328	329
13_	0B0	0B1	0B2	0B3	0B4	0B5	0B6	0B7	0B8	0B9		63_	330	331	332	333	334	335	336	337	338	339
14_	0C0	0C1	0C2	0C3	0C4	0C5	0C6	0C7	0C8	0C9		64_	340	341	342	343	344	345	346	347	348	349
15_	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7	0D8	0D9		65_	350	351	352	353	354	355	356	357	358	359
16_	0E0	0E1	0E2	0E3	0E4	0E5	0E6	0E7	0E8	0E9		66_	360	361	362	363	364	365	366	367	368	369
17_	0F0	0F1	0F2	0F3	0F4	0F5	0F6	0F7	0F8	0F9		67_	370	371	372	373	374	375	376	377	378	379
18_	08A	08B	0AA	0AB	0CA	0CB	0EA	0EB	0CE	0CF		68_	30A	30B	32A	32B	34A	34B	36A	36B	34E	34F
19_	09A	09B	0BA	0BB	0DA	0DB	0FA	0FB	0DE	0DF		69_	31A	31B	33A	33B	35A	35B	37A	37B	35E	35F
20_	100	101	102	103	104	105	106	107	108	109		70_	380	381	382	383	384	385	386	387	388	389
21_	110	111	112	113	114	115	116	117	118	119		71_	390	391	392	393	394	395	396	397	398	399
22_	120	121	122	123	124	125	126	127	128	129		72_	3A0	3A1	3A2	3A3	3A4	3A5	3A6	3A7	3A8	3A9
23_	130	131	132	133	134	135	136	137	138	139		73_	3B0	3B1	3B2	3B3	3B4	3B5	3B6	3B7	3B8	3B9
24_	140	141	142	143	144	145	146	147	148	149		74_	3C0	3C1	3C2	3C3	3C4	3C5	3C6	3C7	3C8	3C9
25_	150	151	152	153	154	155	156	157	158	159		75_	3D0	3D1	3D2	3D3	3D4	3D5	3D6	3D7	3D8	3D9
26_	160	161	162	163	164	165	166	167	168	169		76_	3E0	3E1	3E2	3E3	3E4	3E5	3E6	3E7	3E8	3E9
27_	170	171	172	173	174	175	176	177	178	179		77_	3F0	3F1	3F2	3F3	3F4	3F5	3F6	3F7	3F8	3F9
28_	10A	10B	12A	12B	14A	14B	16A	16B	14E	14F		78_	38A	38B	3AA	3AB	3CA	3CB	3EA	3EB	3CE	3CF
29_	11A	11B	13A	13B	15A	15B	17A	17B	15E	15F		79_	39A	39B	3BA	3BB	3DA	3DB	3FA	3FB	3DE	3DF
30_	180	181	182	183	184	185	186	187	188	189		80_	00C	00D	10C	10D	20C	20D	30C	30D	02E	02F
31_	190	191	192	193	194	195	196	197	198	199		81_	01C	01D	11C	11D	21C	21D	31C	31D	03E	03F
32_	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A7	1A8	1A9		82_	02C	02D	12C	12D	22C	22D	32C	32D	12E	12F
33_	1B0	1B1	1B2	1B3	1B4	1B5	1B6	1B7	1B8	1B9		83_	03C	03D	13C	13D	23C	23D	33C	33D	13E	13F
34_	1C0	1C1	1C2	1C3	1C4	1C5	1C6	1C7	1C8	1C9		84_	04C	04D	14C	14D	24C	24D	34C	34D	22E	22F
35_	1D0	1D1	1D2	1D3	1D4	1D5	1D6	1D7	1D8	1D9		85_	05C	05D	15C	15D	25C	25D	35C	35D	23E	23F
36_	1E0	1E1	1E2	1E3	1E4	1E5	1E6	1E7	1E8	1E9		86_	06C	06D	16C	16D	26C	26D	36C	36D	32E	32F
37_	1F0	1F1	1F2	1F3	1F4	1F5	1F6	1F7	1F8	1F9		87_	07C	07D	17C	17D	27C	27D	37C	37D	33E	33F
38_	18A	18B	1AA	1AB	1CA	1CB	1EA	1EB	1CE	1CF		88_	00E	00F	10E	10F	20E	20F	30E	30F	06E	06F
39_	19A	19B	1BA	1BB	1DA	1DB	1FA	1FB	1DE	1DF		89_	01E	01F	11E	11F	21E	21F	31E	31F	07E	07F
40_	200	201	202	203	204	205	206	207	208	209		90_	08C	08D	18C	18D	28C	28D	38C	38D	0AE	0AF
41_	210	211	212	213	214	215	216	217	218	219		91_	09C	09D	19C	19D	29C	29D	39C	39D	0BE	0BF
42_	220	221	222	223	224	225	226	227	228	229		92_	0AC	0AD	1AC	1AD	2AC	2AD	3AC	3AD	1AE	1AF
43_	230	231	232	233	234	235	236	237	238	239		93_	0BC	0BD	1BC	1BD	2BC	2BD	3BC	3BD	1BE	1BF
44_	240	241	242	243	244	245	246	247	248	249		94_	0CC	0CD	1CC	1CD	2CC	2CD	3CC	3CD	2AE	2AF
45_	250	251	252	253	254	255	256	257	258	259		95_	0DC	0DD	1DC	1DD	2DC	2DD	3DC	3DD	2BE	2BF
46_	260	261	262	263	264	265	266	267	268	269		96_	0EC	0ED	1EC	1ED	2EC	2ED	3EC	3ED	3AE	3AF
47_	270	271	272	273	274	275	276	277	278	279		97_	0FC	0FD	1FC	1FD	2FC	2FD	3FC	3FD	3BE	3BF
48_	20A	20B	22A	22B	24A	24B	26A	26B	24E	24F		98_	08E	08F	18E	18F	28E	28F	38E	38F	0EE	0EF
49_	21A	21B	23A	23B	25A	25B	27A	27B	25E	25F		99_	09E	09F	19E	19F	29E	29F	39E	39F	0FE	0FF

Figure 469. BCD-to-DPD encodings

The following table shows how declets are converted to groups of three BCD digits. The parenthesized values in the table are non-preferred encodings; they are accepted as input values, but are not generated by any arithmetic operation.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00_	000	001	002	003	004	005	006	007	008	009	080	081	800	801	880	881
01_	010	011	012	013	014	015	016	017	018	019	090	091	810	811	890	891
02_	020	021	022	023	024	025	026	027	028	029	082	083	820	821	808	809
03_	030	031	032	033	034	035	036	037	038	039	092	093	830	831	818	819
04_	040	041	042	043	044	045	046	047	048	049	084	085	840	841	088	089
05_	050	051	052	053	054	055	056	057	058	059	094	095	850	851	098	099
06_	060	061	062	063	064	065	066	067	068	069	086	087	860	861	888	889
07_	070	071	072	073	074	075	076	077	078	079	096	097	870	871	898	899
08_	100	101	102	103	104	105	106	107	108	109	180	181	900	901	980	981
09_	110	111	112	113	114	115	116	117	118	119	190	191	910	911	990	991
0A_	120	121	122	123	124	125	126	127	128	129	182	183	920	921	908	909
0B_	130	131	132	133	134	135	136	137	138	139	192	193	930	931	918	919
0C_	140	141	142	143	144	145	146	147	148	149	184	185	940	941	188	189
0D_	150	151	152	153	154	155	156	157	158	159	194	195	950	951	198	199
0E_	160	161	162	163	164	165	166	167	168	169	186	187	960	961	988	989
0F_	170	171	172	173	174	175	176	177	178	179	196	197	970	971	998	999
10_	200	201	202	203	204	205	206	207	208	209	280	281	802	803	882	883
11_	210	211	212	213	214	215	216	217	218	219	290	291	812	813	892	893
12_	220	221	222	223	224	225	226	227	228	229	282	283	822	823	828	829
13_	230	231	232	233	234	235	236	237	238	239	292	293	832	833	838	839
14_	240	241	242	243	244	245	246	247	248	249	284	285	842	843	288	289
15_	250	251	252	253	254	255	256	257	258	259	294	295	852	853	298	299
16_	260	261	262	263	264	265	266	267	268	269	286	287	862	863	(888)	(889)
17_	270	271	272	273	274	275	276	277	278	279	296	297	872	873	(898)	(899)
18_	300	301	302	303	304	305	306	307	308	309	380	381	902	903	982	983
19_	310	311	312	313	314	315	316	317	318	319	390	391	912	913	992	993
1A_	320	321	322	323	324	325	326	327	328	329	382	383	922	923	928	929
1B_	330	331	332	333	334	335	336	337	338	339	392	393	932	933	938	939
1C_	340	341	342	343	344	345	346	347	348	349	384	385	942	943	388	389
1D_	350	351	352	353	354	355	356	357	358	359	394	395	952	953	398	399
1E_	360	361	362	363	364	365	366	367	368	369	386	387	962	963	(988)	(989)
1F_	370	371	372	373	374	375	376	377	378	379	396	397	972	973	(998)	(999)
20_	400	401	402	403	404	405	406	407	408	409	480	481	804	805	884	885
21_	410	411	412	413	414	415	416	417	418	419	490	491	814	815	894	895
22_	420	421	422	423	424	425	426	427	428	429	482	483	824	825	848	849
23_	430	431	432	433	434	435	436	437	438	439	492	493	834	835	858	859
24_	440	441	442	443	444	445	446	447	448	449	484	485	844	845	488	489
25_	450	451	452	453	454	455	456	457	458	459	494	495	854	855	498	499
26_	460	461	462	463	464	465	466	467	468	469	486	487	864	865	(888)	(889)
27_	470	471	472	473	474	475	476	477	478	479	496	497	874	875	(898)	(899)
28_	500	501	502	503	504	505	506	507	508	509	580	581	904	905	984	985
29_	510	511	512	513	514	515	516	517	518	519	590	591	914	915	994	995
2A_	520	521	522	523	524	525	526	527	528	529	582	583	924	925	948	949
2B_	530	531	532	533	534	535	536	537	538	539	592	593	934	935	958	959
2C_	540	541	542	543	544	545	546	547	548	549	584	585	944	945	588	589
2D_	550	551	552	553	554	555	556	557	558	559	594	595	954	955	598	599
2E_	560	561	562	563	564	565	566	567	568	569	586	587	964	965	(988)	(989)
2F_	570	571	572	573	574	575	576	577	578	579	596	597	974	975	(998)	(999)
30_	600	601	602	603	604	605	606	607	608	609	680	681	806	807	886	887
31_	610	611	612	613	614	615	616	617	618	619	690	691	816	817	896	897
32_	620	621	622	623	624	625	626	627	628	629	682	683	826	827	868	869
33_	630	631	632	633	634	635	636	637	638	639	692	693	836	837	878	879
34_	640	641	642	643	644	645	646	647	648	649	684	685	846	847	688	689
35_	650	651	652	653	654	655	656	657	658	659	694	695	856	857	698	699
36_	660	661	662	663	664	665	666	667	668	669	686	687	866	867	(888)	(889)
37_	670	671	672	673	674	675	676	677	678	679	696	697	876	877	(898)	(899)
38_	700	701	702	703	704	705	706	707	708	709	780	781	906	907	986	987
39_	710	711	712	713	714	715	716	717	718	719	790	791	916	917	996	997
3A_	720	721	722	723	724	725	726	727	728	729	782	783	926	927	968	969
3B_	730	731	732	733	734	735	736	737	738	739	792	793	936	937	978	979
3C_	740	741	742	743	744	745	746	747	748	749	784	785	946	947	788	789
3D_	750	751	752	753	754	755	756	757	758	759	794	795	956	957	798	799
3E_	760	761	762	763	764	765	766	767	768	769	786	787	966	967	(988)	(989)
3F_	770	771	772	773	774	775	776	777	778	779	796	797	976	977	(998)	(999)

Figure 470. DPD-to-BCD translation

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
ADTR	B3D2
ADTRA	B3D2
AXTR	B3DA
AXTRA	B3DA
CDFTR	B351
CDGTR	B3F1
CDGTRA	B3F1
CDLFTR	B353
CDLGTR	B952
CDSTR	B3F3
CDTR	B3E4
CDUTR	B3F2
CDZT	EDAA
CEDTR	B3F4
CEXTR	B3FC
CFDTR	B941
CFXTR	B949
CGDTR	B3E1
CGDTRA	B3E1
CGXTR	B3E9
CGXTRA	B3E9
CLFDTR	B943
CLFXTR	B94B
CLGXTR	B94A
CLGDTR	B942
CPSDR	B372
CSDTR	B3E3
CSXTR	B3EB
CUDTR	B3E2
CUXTR	B3EA

Mnemonic	Opcode
CXFTR	B359
CXGTR	B3F9
CXGTRA	B3F9
CXLFTR	B95B
CXLGTR	B95A
CXSTR	B3FB
CXTR	B3EC
CXUTR	B3FA
CXZT	EDAB
CZDT	EDA8
CZXT	EDA9
DDTR	B3D1
DDTRA	B3D1
DXTR	B3D9
DXTRA	B3D9
EEDTR	B3E5
EEXTR	B3ED
ESDTR	B3E7
ESXTR	B3EF
FIDTR	B3D7
FIXTR	B3DF
IEDTR	B3F6
IEXTR	B3FE
KDTR	B3E0
KXTR	B3E8
LCDFR	B373
LDETR	B3D4
LDGR	B3C1
LDXTR	B3DD
LEDTR	B3D5

Mnemonic	Opcode
LGDR	B3CD
LNDFR	B371
LPDFR	B370
LTDTR	B3D6
LTXTR	B3DE
LXDTR	B3DC
MDTR	B3D0
MDTRA	B3D0
MXTR	B3D8
MXTRA	B3D8
QADTR	B3F5
QAXTR	B3FD
RRDTR	B3F7
RRXTR	B3FF
SDTR	B3D3
SDTRA	B3D3
SLDT	ED40
SLXT	ED48
SRDT	ED41
SRNMT	B2B9
SRXT	ED49
SXTR	B3DB
SXTRA	B3DB
TDCDT	ED54
TDCET	ED50
TDCXT	ED58
TDGDT	ED55
TDGET	ED51
TDGXT	ED59

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
B2B9	SRNMT
B3CD	LGDR
B3C1	LDGR
B3DA	AXTR
B3DA	AXTRA
B3DB	SXTR
B3DB	SXTRA
B3DC	LXDTR
B3DD	LDXTR
B3DE	LTXTR
B3DF	FIXTR
B3D0	MDTR
B3D0	MDTRA
B3D1	DDTR
B3D1	DDTRA
B3D2	ADTR
B3D2	ADTRA
B3D3	SDTR
B3D3	SDTRA
B3D4	LDETR
B3D5	LEDTR
B3D6	LTDTR
B3D7	FIDTR
B3D8	MXTR
B3D8	MXTRA
B3D9	DXTR
B3D9	DXTRA
B3EA	CUXTR
B3EB	CSXTR
B3EC	CXTR

Opcode	Mnemonic
B3ED	EEXTR
B3EF	ESXTR
B3E0	KDTR
B3E1	CGDTR
B3E1	CGDTRA
B3E2	CUDTR
B3E3	CSDTR
B3E4	CDTR
B3E5	EEDTR
B3E7	ESDTR
B3E8	KXTR
B3E9	CGXTR
B3E9	CGXTRA
B3FA	CXUTR
B3FB	CXSTR
B3FC	CEXTR
B3FD	QAXTR
B3FE	IEXTR
B3FF	RRXTR
B3F1	CDGTR
B3F1	CDGTRA
B3F2	CDUTR
B3F3	CDSTR
B3F4	CEDTR
B3F5	QADTR
B3F6	IEDTR
B3F7	RRDTR
B3F9	CXGTR
B3F9	CXGTRA
B351	CDFTR

Opcode	Mnemonic
B353	CDLFTR
B359	CXFTR
B370	LPDFR
B371	LNDFR
B372	CPSDR
B373	LCDFR
B94A	CLGXTR
B94B	CLFXTR
B941	CFDTR
B942	CLGDTR
B943	CLFDTR
B949	CFXTR
B95A	CXLGTR
B95B	CXLFTR
B952	CDLGTR
ED40	SLDT
ED41	SRDT
ED48	SLXT
ED49	SRXT
ED50	TDCET
ED51	TDGET
ED54	TDCDT
ED55	TDGDT
ED58	TDCXT
ED59	TDGXT
EDA8	CZDT
EDA9	CZXT
EDAA	CDZT
EDAB	CXZT

Exercises

35.15.1.(3) Several decimal floating-point instructions have an M_4 mask field. Make a table showing (a) their possible values, (b) the type or types of instructions to which each value applies, and (c) the effect of each value for the affected instructions.

Terms and Definitions

cohort
In a given format, a set of decimal floating-point numbers having the same numeric value but different quanta.

declet
A 10-bit encoding of three Binary Coded Decimal (BCD) digits. Declets may have two forms:

- *canonical*: preferred (and generated) values, and
- *non-canonical*: any of 24 non-preferred encodings accepted as operands, but not generated by any arithmetic operation.

DPD
Densely Packed Decimal; an encoding into declets.

extreme exponent
An exponent with maximum positive or negative value.

payload
Diagnostic information contained in a the significand of a NaN.

preferred exponent
The exponent of the result of every numeric operation has a preferred value, either that of one of the operands or a value providing the maximum number of significant digits.

preferred quantum
The quantum selected for the result of an operation that maximizes the number of significant digits, including low-order zero digits. Equivalent to *preferred exponent*.

quantum
The value of a unit in the low-order digit of a decimal floating-point number.

Programming Problems

Problem 35.1.(3)+ Write a program that reads records containing eight hex digits representing a short decimal floating-point number. Then, display the value of the number in "scientific" format: `sd.ddddddEsdd` where `s` is a sign, and `d` represents decimal digits. If the hex digits represent a special value, print its sign followed by `Inf`, `QNaN`, or `SNaN` as appropriate.

Problem 35.2.(4) In decimal floating-point arithmetic, the equation $(1.0/N)\times N=1.0$ is true for some values of N between 1 and 100 and not for others. Write a program that tests this relation in long and extended precision decimal floating-point arithmetic. Show the values of N for which the relation fails for both lengths, and for which the relation fails for only one of the lengths.

For your own edification and/or extra credit:

- For those values of N for which the equation fails, determine the relative difference between 1.0 and the result of $(1.0/N)\times N$.
- What rounding modes should be used for your solution? How would other choices affect the results?
- What property is shared by all the values that fail?

Problem 35.3.(3)+ Write a subroutine named **`DPD2BCD`** that converts a declet to the decimal equivalent 3 hex digits that represent the value of the declet.

Then, write a driver program that will call **`DPD2BCD`** for each of the 1024 possible bit combinations, and display in tabular form (a) the decimal value of the declet, (b) the declet as three hexadecimal digits, and (c) the decimal number between 0 and 999 represented by the declet.

36. Floating-Point Summary

```
 3333333333    6666666666
333333333333  666666666666
33        33  66        66
          33  66
          33  66
        3333  66666666666
        3333  666666666666
          33  66        66
          33  66        66
33        33  66        66
333333333333  666666666666
 3333333333    6666666666
```

This chapter has covered a wide range of topics; this final section summarizes properties of floating-point data and arithmetic, and provides some general observations and reminders.

36.1. Floating-Point Data Representations

The three System z floating-point representations are summarized in Table 387 for short, long, and extended precision data respectively.

Table 387 (Page 1 of 2). Summary of System z floating-point representations

Property	**Hexadecimal**	**Binary**	**Decimal**
Operand width, in bits	32, 64, 128	32, 64, 128	32, 64, 128
Base/radix	16	2	10
Representation of significand digits	Fraction	Fraction	Integer or fraction
Significant digits	All are present	Implicit high-order bit for normalized data	All are present; high-order digit is part of Combination Field
Significand width, in radix digits	6, 14, 28	24, 53, 113	7, 16, 34
Maximum equivalent decimal precision	6, 15, 32	6, 15, 33	7, 16, 34
Exponent width, in bits	Always 7	8, 11, 15	8, 10, 14
Exponent representation	Binary	Binary	Binary
Exponent sign	In biased exponent	In biased exponent	In biased exponent
Exponent bias	64	127, 1023, 16383	101, 398, 6176
Maximum exponent	+63	+127, +1023, +16383	+96, +384, +6144
Minimum exponent	−64	−126, −1022, −16382	−95, −383, −6143

Table 387 (Page 2 of 2). Summary of System z floating-point representations			
Property	**Hexadecimal**	**Binary**	**Decimal**
Maximum normalized value	$7.2\times10^{+75}$	$3.4\times10^{+38}$ $1.8\times10^{+308}$ $1.2\times10^{+4392}$	10^{+97} 10^{+385} 10^{+6145}
Minimum normalized value	7.2×10^{-79}	1.2×10^{-38} 2.2×10^{-308} 3.4×10^{-4392}	10^{-95} 10^{-383} 10^{-6143}
Minimum denormalized value	5.1×10^{-85} 1.2×10^{-94} 1.7×10^{-111}	1.4×10^{-45} 4.9×10^{-324} 6.5×10^{-4966}	10^{-101} 10^{-398} 10^{-6176}
Scale factor for overflow/underflow exponent wrap	none	192 1536 24576	— 576 9216
Scale factor for “Load Rounded” exponent wrap	none	— 512 8192	— 192 3072
Special values	None	QNaN, SNaN, Infinity	QNaN, SNaN, Infinity
Multiple representations of a value	Yes	No	Yes
Unnormalized values	Yes	Yes (for tiny values)	Yes
Rounding options for arithmetic	None	4 modes	8 modes
Sign of zero arithmetically significant	No	Yes	Yes

Each representation has advantages and disadvantages:

- Hexadecimal floating-point has a fixed-width characteristic, so it's easier to extract the significand and to extend or shorten operands; it supports redundant representations.
- Binary floating-point has no redundant values or representations, but lengthening or shortening operands requires special instructions.
- Decimal floating-point has a very complex representation and supports redundant values, but its arithmetic is more intuitive than hexadecimal or binary.
- Precisions have minor differences:
  - Short binary has 0 to 3 more fraction bits than short hexadecimal
  - Long binary has 0 to 3 *fewer* fraction bits than long hexadecimal
  - Extended IEEE binary has 1 to 4 more fraction bits than extended hexadecimal
- Decimal floating-point has greater decimal precision than either hexadecimal or binary floating-point, for all operand lengths.

Exercises

36.1.1.(1) What possibilities must you consider in converting short-precision floating-point data between hexadecimal and binary representations?

36.1.2.(1) What possibilities must you consider in converting long-precision floating-point data between hexadecimal and binary representations?

36.1.3.(1) What possibilities must you consider in converting extended-precision floating-point data between hexadecimal and binary representations?

36.2. Floating-Point Properties

Because precision and range are limited, the fundamental behaviors of floating-point are always present; and, some numbers cannot be represented exactly. For example, consider the decimal value `0.1`: its value in binary is `0.0001 1001 1001 1001 1001 1001 1001 1001 ...`, an unbounded bit string. When representing 0.1 as a normalized short hexadecimal floating-point value, the fraction has 21 significant bits: `X'19999A'`, with relative error $\approx 2^{-22}$. When represented as a short precision binary floating-point value, the fraction has 24 significant bits: `X'CCCCCD'`, with relative error $\approx 2^{-25}$.

This representation error may lead to unexpected results. Suppose we add 0.1 to itself 8 times, and multiply 0.1 by 8, using short binary and hexadecimal arithmetic. The results are shown in Table 388, where the differences from the decimal result are due to the imprecise representation of 0.1 in hexadecimal and binary.

Operation	Binary	Hexadecimal	Decimal
Add 0.1 8 times	.8000000<u>715</u>...	.8000000<u>119</u>...	.8000000
Multiply 0.1 by 8	.8000000<u>119</u>...	.8000000<u>119</u>...	.8000000

Table 388. Adding 0.1 in hexadecimal, binary, and decimal floating-point

This repeating-fraction problem doesn't go away when you use decimal floating-point: it cannot precisely represent fractions like 2/3 and 1/7.

Exercises

36.2.1.(2)+ Using 6-digit decimal floating-point numbers and arithmetic (not a type used in System z!), add 2/3 to 1.00000 3 times, with rounding to the nearest 6-digit value at each step. If you start (a) with 2/3 = 0.666667 and (b) with 2/3 = 0.666666, what are the results?

36.2.2.(2)+ Using the same 6-digit decimal floating-point numbers as in Exercise 36.2.1, add three copies of each representation of 2/3 to itself, and then add 1.00000. What are the results?

36.3. Floating-Point Exceptions

Hexadecimal and binary/decimal exceptions have similarities *and* differences, as summarized in Tables 389 and 390.

Exception	Maskable	Masked result	Interrupt action/result
Exponent overflow	No	—	Scaled exponent
Exponent underflow	Yes	True zero	Scaled exponent
Zero divide	No	—	Dividend operand unchanged
Lost significance	Yes	True zero	Pseudo-zero

Table 389. Exception behavior for hexadecimal floating-point

Exception	Maskable	Masked result	Interrupt action/result
Invalid operation	Yes	QNaN	Operation suppressed
Zero divide	Yes	Signed infinity	Operation suppressed
Exponent overflow	Yes	Infinity or MaxReal	Scaled exponent
Exponent underflow	Yes	Zero (or BFP de-norm)	Scaled exponent
Inexact result	Yes	Calculated result	Calculated result
Quantum exception (DFP only)	Yes	Calculated result	Calculated result

Table 390. Exception behavior for binary and decimal floating-point

Differences to consider

Both hexadecimal and IEEE BFP/DFP results can differ significantly between masked and unmasked actions.

36.4. Defining Floating-Point Constants

Each type allows you to specify a length modifier, as shown in Table 391. The hexadecimal and binary types can be generated at less than their default lengths, but decimal floating-point constants must have fixed lengths even if they are not aligned on default boundaries. (Bit-length modifiers were described in Section 17.5 on page 257.)

Constant Type	Default Length	Length Modifier Range
E,EH (hexadecimal)	4	.12 to 8
EB (binary)	4	.9 to 4
ED (decimal)	4	.32 or 4 only
D,DH (hexadecimal)	8	.12 to 8
DB (binary)	8	.12 to 8
DD (decimal	8	.64 or 16 only
L,LH,LQ (hexadecimal)	8	.12 to 16
LB (binary)	16	.16 to 16
LD (decimal)	16	.128 or 16 only

Table 391. Length modifiers of floating-point constants

Table 392 summarizes the rounding-mode suffixes you can use when defining floating-point constants. Values less than 8 are used for hexadecimal and binary constants. and values greater than or equal to 8 are used for decimal floating-point constants.

Mode	Rn	Description
1	R1	Round half-up (to nearest, ties away from 0) (HFP default)
4	R4	Round half-even (to nearest, ties to nearest even) (BFP Default)
5	R5	Round toward zero (truncate)
6	R6	Round to $+\infty$; if $-$, truncate
7	R7	Round to $-\infty$; if $+$, truncate
8	R8	Round half-even (to nearest, ties to nearest even) (default)
9	R9	Round toward zero (truncate)
10	R10	Round to $+\infty$; if $-$, truncate
11	R11	Round to $-\infty$; if $+$, truncate
12	R12	Round half-up (to nearest, ties away from 0)
13	R13	Round half-down (to nearest, ties toward 0)
14	R14	Round up (away from zero)
15	R15	Round for reround (“prepare for shorter precision”)

Table 392. Assembler rounding-mode suffixes for floating-point constants

Exercises

36.4.1.(2) Table 391 indicates that the minimum bit length for hexadecimal floating-point constants is 12 bits. However,

```
DC    EL.9'8'
```

is quite acceptable to the Assembler. Why then is the stated minimum bit length 12?

36.4.2.(3)+ Consider the representations of the decimal constant 0.1: Its short binary floating-point representation is X'3DCCCCCD' and its short decimal floating-point representation is X'22400001'. What are the values of these two words if they are mistakenly interpreted as short *hexadecimal* floating-point constants?

36.4.3.(3)+ The short hexadecimal floating-point representation of 0.1 is X'4019999A'. What would be the value of this constant if treated as if it is binary and decimal floating-point?

36.4.4.(3)+ The short decimal floating-point representation of 1 is X'22500001'. What would be the value of this constant if treated as if it is hexadecimal and binary floating-point?

36.4.5.(3)+ These four floating-point constants were found in a program that didn't document their type. What values would they take if interpreted as short hexadecimal, binary, and decimal floating-point constants?

(1) X'3F800000'
(2) X'41100000'
(3) X'42640000'
(4) X'7FFFFFFF'

36.5. Converting Among Decimal, Hexadecimal and Binary Representations

When you display the values of floating-point numbers, they will almost always be wanted in decimal form. For decimal floating-point, you only need to format the results; no conversion is needed because values are already in decimal format. For the other two representations, however, the data must be converted from a binary format (including hexadecimal) to decimal. These conversions can be very difficult to do correctly: approximate conversions between decimal and hex or binary are easy, but some common values require great care. We'll discuss this problem in two forms, known as “In-Out” and “Out-In” conversions.

An “In” conversion converts data from decimal to a machine radix (hexadecimal or binary) and an “Out” conversion converts data from a from machine radix to decimal. The conversions between decimal and binary *must* be correctly rounded for all values[237] for these rules to apply.

36.5.1. In-Out Conversions

When we convert external data from a decimal representation to internal hexadecimal or binary floating-point, do some computations, and then convert the internal floating-point values back to decimal for display or printing, we have done an In-Out conversion. To be assured that the results are reliable, we must answer this question:

- If the precision of the external decimal data is D digits, what internal precision p is required to be sure we can *correctly* represent the decimal value (the “In” conversion); and when converted back to decimal (the “Out” conversion), can we *correctly* recover all D decimal digits?

This observation helps define the term “equivalent decimal digits”. Using a floating-point system FP(r,p) with radix r and p base-r digits, if

1. $10^J \neq r^K$ for any nonzero J and K (that is, no power of 2 is a power of 10, which is true for both hexadecimal and binary), and
2. all D-digit decimal floating-point numbers can be correctly converted *to* a member of FP(r,p), and when converted *back* to decimal will correctly recover the original decimal value, and

[237] Today, these conversions are almost always correctly rounded, but for many years their complexity was not well understood.

3. conversions are exact (that is, deliver the nearest neighbor of the infinitely precise value), and are correctly rounded.

then we say that FP(r,p) can *faithfully represent* D-digit decimal numbers.

Starting with decimal data with precision D, the required internal hexadecimal and binary floating-point precisions for faithful conversion are summarized in Table 393.

Precision	Decimal Digits	Hexadecimal Digits	Binary Digits
Short	6	6	21
Long	15	14	51
Extended	32 33	28 —	— 111

Table 393. Internal precision required for faithful In-Out conversion

For example, suppose the decimal data has precision 6 digits. Faithful conversion to hexadecimal or binary requires 6 hexadecimal digits or 21 binary digits, so decimal data with 6 digits is faithfully represented in both short floating-point representations. But if the decimal data has 7 digits, the number of internal hexadecimal and binary digits is too small, so seven-digit decimal numbers cannot be faithfully represented in short floating-point. Thus we say that FP(16,6) and FP(2,24) faithfully represent 6 decimal digits. This result may not be what you would expect from calculating the weight of the low-order bit of a floating-point number. In short hexadecimal floating-point, even though $16^{-6} \approx 0.59\times10^{-7}$, all 6-digit decimal floating-point values but not all 7-digit decimal floating-point values can can be converted to hexadecimal and back with complete recovery.

Similar considerations apply to long floating-point: at most 15 decimal digits can be faithfully represented in hexadecimal and binary. For extended precision, the two representations behave differently: hexadecimal floating-point can faithfully represent 32 decimal digits, while binary floating-point can faithfully represent 33.

36.5.2. Out-In Conversions

For Out-In conversions, we start with internal floating-point data, convert it to external decimal format, and then want to convert it back to internal format without losing the precision of the original internal values. This means we must know how many decimal digits are required; they are shown in Table 394.

Precision	Binary Digits	Hexadecimal Digits	Decimal Digits
Short	24	6	9
Long	53 —	— 14	17 18
Extended	— 113	28 —	35 36

Table 394. Decimal precision required for faithful Out-In conversion

For example, to recover *long* internal floating-point values converted to decimal and back,

- long binary requires 17 decimal digits for faithful representation of the internal value,
- long hexadecimal requires 18 decimal digits,
- and (as for In-Out conversions) correct conversion is required.

This is a useful “Rule of Thumb” for In-Out and Out-In conversions:

In-Out and Out-In Conversions

Out-In conversions need 3 more decimal digits than In-Out.

36.5.3. The PFPO Instruction (*)

The System z PFPO instruction converts floating-point data formats among one another.

Op	Mnem	Type	Instruction
010A	PFPO	E	Perform Floating-Point Operation

Table 395. Perform Floating-Point Operation instruction

Its assembler instruction statement has no operands! The floating-point source and target operands are in FPR4 and FPR0 respectively (with their paired registers for extended-precision operands). GR0 is set to a complex bit pattern describing the format of the source and target operands, and GR1 contains the return code. For example, to convert a short BFP operand in FPR4 to a long HFP operand in FPR0, you could write

```
        LE    4,=EB'3.141592645' Source operand in FPR4, X'40490FDB'
        L     0,=X'01010501'     Convert according to current BFP mode
        PFPO  ,                  Convert short BFP to long HFP
        STD   0,HFPLong          Store result, X'413243F6C0000000'
        LTR   1,1                Check return code
        JNZ   WhatNext           Do something if nonzero
        - - -
HFPLong DS    D                  Converted HFP result
```

The instruction handles many operand combinations; for details, see the *z/Architecture Principles of Operation*.

Exercises

36.5.1.(2)+ Write a short Assembler Language program defining hexadecimal floating-point constants for the six 7-digit decimal floating-point values 0.625000xE−1, where x takes values from 3 to 8. What hexadecimal floating-point values are generated?

36.5.2.(3) If the short hexadecimal floating-point number X'40100001' is converted to 7-digit decimal, what is the result?

36.6. “Real” and “Realistic” (Floating-Point) Arithmetic

Computation uses actual numbers, not mathematical real-number abstractions, so we must cope with problems of precision, rounding, significance, etc. The mathematician's “real” numbers provide elegant analyses, but don't always describe the “realistic” world where weights, dimensions, and other values are known only imprecisely. The quantities used for computation necessarily have finite precision and errors of measurement or estimation, and computational techniques have finite precision.

The abstract formulation of a problem can help define the form of a computation, but it's important to remember that usable results may depend on knowing the behavior of the adaptations you make in order to calculate meaningful results.

The “laws” of real arithmetic such as associativity, distributivity, etc. apply only to abstract numbers and in limited ways to “realistic” numbers, so you may sometimes unknowingly make unsafe assumptions when you write your programs. We'll compare “real” and “realistic” laws in Table 396 on page 746, where real numbers are represented by lower case letters (a,b), and floating-point numbers are represented by capital letters (A,B).[238] The letter “r” denotes the radix of the representation.

[238] A useful reference is *Floating-Point Computation* by Pat H. Sterbenz, Prentice-Hall, 1974.

Table 396. Laws of real and realistic arithmetic		
"Law"	**Arithmetic with "real" numbers**	**Arithmetic with "realistic" numbers**
Closure	The product and sum of real numbers a and b are reals.	The product (AB) and sum (A+B) of the floating-point numbers A and B may not exist in FPF(r,p). Example: FPF(16,6) (HFP) contains 90009.0 and 0.84375, but not their sum, difference, quotient, or product.
Commutative	a+b = b+a, ab = ba	True for System z and *almost* all other systems.
Associative Addition	(a+b)+c = a+(b+c)	Fails. If A, B, and C have the same sign, the results may differ by r ulps.
Associative Multiplication	(ab)c = a(bc)	Fails. If both products are in range, the results may differ by 2r ulps (fails spectacularly if exponent spills occur).
Distributive	a(b+c) = ab+bc	Fails. If B and C have the same sign, then the results differ by at most r ulps.
Additive Unit	There is a real number 0 such that a+0 = 0+a = a	True for most systems (unless underflow intrudes).
Multiplicative Unit	There is a real number 1 such that a×1 = 1×a = a	True for System z and *almost* all other systems.
Additive Inverse	For any real a there is a real −a with a+(−a)=(−a)+a=0	True in all known systems.
Multiplicative Inverse	For any real a, if a≠0 there is a real a^{-1} with $a \times a^{-1} = a^{-1} \times a = 1$	Fails to hold in many systems; there may be no representable inverses. (System z hexadecimal floating-point has no inverse for any magnitude ≤X'02100000'=16^{-63}). • Several A values may have the same A^{-1} (=1÷A) • Some A^{-1} values give $A \times A^{-1} \neq 1$ – In HFP, A×(1/A) may differ from 1 by 16 ulps. • If r > 2, a number with fraction $1-r^{-p}$ has r−1 inverses in FPF(r,p) with chopped arithmetic, and approximately (r−1)/2 inverses with rounded arithmetic – In HFP, $A^{-1} = B^{-1}$ means only that A and B may differ by 16 ulps • In HFP, numbers with fraction $1-n \times r^{-p}$ have inverse r^{-1} if 1≤ n≤ 15
Zero Divisor	If ab = 0, then at least one of a or b must vanish	Frequently fails if overflow or underflow occurs.
Cancellation	1. b+(a−b)=a 2. a×(b/a)=b 3. If ab = ac and a ≠ 0, then b=c	1. Fails, especially in the presence of underflow or overflow. 2. Fails. 3. Fails. B and C may differ by r−1 ulps.
Division	If b ≠ 0, and a/b means ab^{-1}, then b(a/b) = a	Fails. If B(A/B) = C, then |A| and |C| may differ by r ulps.
Subtraction	If (a−b) means a+(−b), then (a+b)−b = a	Frequently fails near underflow threshholds, or if |B| greatly exceeds |A|.
Inequalities	1. a < b 2. If a < b then for all c, a+c < b+c 3. If a < b and c < d then a+c < b+d 4. If b < c and a > 0 then ab < ac	Fails; strict inequalities must be weakened to tolerate equality. 1. A ≤ B 2. A+C ≤ B+C 3. A+C ≤ B+D 4. AB ≤ AC

1. Floating-point numbers are actually a subset of the rational numbers, but they also don't satisfy all the mathematical properties of rationals (which have unbounded numerator and denominator).
2. Neither equalities nor inequalities reliably persist across floating-point operations.
3. Multiplying any fraction by a power of the radix leaves the fraction unchanged, unless there is no guard digit and the product is truncated.
4. Inverses of tiny hex numbers overflow: the inverse of `X'02100001'` = HexMax, but any smaller value causes exponent overflow.
5. An advantage of the binary floating-point representation: inverses exist for *all* normal values.
6. Some operations create meaningless results. Even in well-behaved programs, ordinary algebraic rules may fail:

Calculation	Possible Problems
`0 / X →0 ?`	`Fails for X = 0`
`1 * X →X ?`	`Fails for X = NaN`
`0 * X →0 ?`	`Fails for X = NaN or infinity`
`X - X →0 ?`	`Fails for X = NaN or infinity`
`X / X →1 ?`	`Fails for X = NaN or infinity`
`X = X →TRUE ?`	`Fails for X = NaN`
`X > Y →Y < X ?`	`Fails for X = NaN`
`-X = 0 - X ?`	`Fails for X = +0`

This table shows that NaNs and infinity violate many common arithmetic “laws”.

Numerical analysis is a “realistic” discipline, coping with realistic problems in a world of realistic numbers; the “real” analysis of advanced mathematics is a helpful but occasionally misleading abstraction because it requires “unrealistic” conditions such as infinite precision and unbounded magnitudes.

Exercises

36.6.1.(2)+ Table 396 on page 746 indicates that the “additive unit” is zero. Give an example of a nonzero hexadecimal floating-point value A such that $A+0 \neq A$.

36.6.2.(2)+ Table 396 on page 746 indicates that the “additive unit” is zero. Give examples of a nonzero numeric binary and decimal floating-point value A such that $A+0 \neq A$.

36.6.3.(4) Find a short HFP constant A such that $A\times(1/A)$ differs from 1 by at least 15 ulps.

36.6.4.(2)+ In hexadecimal floating-point, what is the value of Max×Min?

36.6.5.(3)+ For each floating-point representation, create an example that shows the failure of additive association.

36.6.6.(3)+ For each floating-point representation, create an example that shows the failure of multiplicative association.

36.6.7.(3)+ For each floating-point representation, create an example that shows the failure of the distributive law.

36.7. When Does Zero Not Behave Like Zero? (*)

Fascinating details

Mathematically, we would expect $X\pm 0=X$, $X\times 0=0$, and $0\div X=0$ (for $X\neq 0$). In floating-point arithmetic, a zero value doesn't always behave like a mathematical zero, especially in addition and subtraction.

36.7.1. Hexadecimal Floating-Point

Hexadecimal floating-point Pseudo-zeros have zero fraction and nonzero characteristic. For example:

Precision	A representative pseudo-zero
Single	X'41000000'
Double	X'41000000 00000000'
Extended	X'41000000 00000000 33000000 00000000'

Table 397. Examples of hexadecimal floating-point pseudo-zeros

If a pseudo-zero is an operand in a multiply operation, or is the dividend in a division operation, the result is a true zero (that is, +0). Thus, multiplication and division are well-behaved mathematically in hexadecimal floating-point arithmetic.

For addition and subtraction, the result depends on the differences between the characteristics of the operands. Figure 471 shows some examples using short-precision operands: the precision of the sum is degraded as the characteristic difference grows.

```
LE    4,=E'123.456'      c(FPR4) = X'427B74BC' = 123.456
LER   0,4                Copy starting value to FPR0
AE    0,=X'45000000'     c(FPR4) = X'427B7400' = 123.453
LER   0,4                Copy starting value to FPR0
AE    0,=X'46000000'     c(FPR4) = X'427B7000' = 123.434
LER   0,4                Copy starting value to FPR0
AE    0,=X'47000000'     c(FPR4) = X'427B0000' = 123.000
LER   0,4                Copy starting value to FPR0
AE    0,=X'48000000'     c(FPR4) = X'42700000' = 112.000
LER   0,4                Copy starting value to FPR0
AE    0,=X'49000000'     c(FPR4) = X'00000000' =   0.0
```

Figure 471. Degraded precision in adding hexadecimal floating-point pseudo-zeros

Because the fraction of the operand with the smaller characteristic is shifted right until the characteristics are equal, more and more significant digits are lost.

- In the first example, the characteristic difference is 3, so the fraction X'7B74BC' is shifted right three places; the digit 4 becomes the guard digit and then appears in the normalized result, so only two digits are lost.
- In the last example, a zero fraction is generated and a significance exception is indicated.

Hexadecimal floating-point pseudo-zeros should be used rarely, and only with care.

36.7.2. Binary Floating-Point

Binary floating-point does not support pseudo-zeros: every zero and nonzero finite number has a unique representation in each precision, except that zero may have either sign. Arithmetic results have the mathematically-expected values:

- Adding zero to or subtracting zero from any operand that is not a NaN produces the original operand. (The sign of an exact zero result depends on the rounding mode.)
- Dividing zero by any finite operand that is not a NaN produces ± 0.
- Multiplying zero by any operand that is not a NaN produces ± 0.
- The square root of zero is a zero with the same sign.
- Operations with $\pm\infty$ behave sensibly.

36.7.3. Decimal Floating-Point

As described in Section 35.3 on page 690, decimal floating-point zeros can have many representations. This can sometimes lead to unintuitive or unexpected results. For example, suppose we add 1 and 0.0000000:

```
LD    4,=DD'0.0000000'   0E-7
LD    2,=DD'1'
ADTR  0,2,4              Sum in FPR0
EEDTR 6,0                Biased exponent in GG6
AGHI  6,-398             Exponent in GR6 = -7
CUDTR 7,0                c(GG7) = X'00000000 10000000'
```

so that the result is $10000000.\times10^{-7}$, with value 1. We see that 1+0=1, but the result in this case has a very different representation than ordinary arithmetic might predict. If the decimal floating-point zero operand had been defined by

```
LD    4,=DD'0.00'
```

and the same instructions were executed, the result would be $100.\times10^{-2} = 1.00$, with value 1 (again!) but with yet another representation.

This behavior isn't limited to zero constants. If the result of an arithmetic operation is zero, we can get similar results:

```
LD    4,=DD'123.456'     c(FPR4)=X'222C0000 00028E56'
SDTR  2,4,4              c(FPR2)=X'222C0000 00000000' (zero)
LD    0,=DD'1'           c(FPR0)=X'22380000 00000001'
ADTR  4,2,0              c(FPR4)=X'222C0000 00000400'
CUDTR 3,4                c(GG3) =C'00000000 00001000'
EEDTR 5,4                Extract biased exponent
AGHI  5,-398             Remove bias; c(GR5)=X'FFFFFFFD'=-3
```

The operands of the SDTR instruction have 3 significant decimal places, so the resulting zero in FPR2 has the same quantum. When 1 is added the quantum is preserved, and the result is $1000\times10^{-3} = 1.000$, with value 1 as expected.

As noted earlier, these results are numerically reasonable, but their representation may not be as easily predictable as with fixed-point binary or decimal arithmetic.

Exercises

36.7.1.(1)+ In a hexadecimal floating-point representation of each precision, how many different pseudo-zeros can be created?

36.7.2.(2) Figure Figure 471 on page 748, what result will be generated if the second operand is X'43000000'? That is, what is the result of this operation:

```
LE    4,=E'123.456'      c(FPR4) = X'427B74BC' = 123.456
AE    4,=X'43000000'     c(FPR4) = ?
```

36.8. Examples of Former Floating-Point Representations and Behaviors (*)

In Table 398 on page 750, a sample of floating-point representations shows the processor, the base (radix) B of the significand, the number of base-B digits in the significand, the equivalent number of decimal digits represented, the exponent width in bits, the instruction format, and the representation of the significand.[239]

[239] To see some other floating-point representations, you may enjoy exploring the web for the formats on other old processors such as the IBM 7030 “Stretch”, the DEC VAX, and the Harris Series 500.

Processor and representation	Base B	Base-B Digits	Equiv. Dec.	Exp. Wid.	Format	Significand Representation
Pr1me 550 series single	2	23	6	8	s/f/e	Two's complement
IEEE binary short precision	2	24	6	8	s/c/f	Sign-magnitude
IBM hex short precision	16	6	6	7	s/c/f	Sign-magnitude
Burroughs 6700 short precision	8	13	10	6	S/s/e/i	Sign-magnitude
CDC 6600/CYBER 70	2	48	14	11	s/e/i	Sign-magnitude
Cray-2 single precision	2	48	14	15	s/e/f	Sign-magnitude
IEEE binary double precision	2	53	15	11	s/c/f	Sign-magnitude
IBM hex double precision	16	14	15	7	s/c/f	Sign-magnitude
IBM hex extended precision	16	28	32	7	s/c/f	Sign-magnitude
Note: s = sign, e = signed exponent, c = characteristic, f = fraction, i = integer.						

Table 398. Examples of other floating-point representations

Most of these representations use a biased characteristic, but the Cray-2 used a signed exponent. Two processors (Burroughs and CDC) used a right-adjusted integer significand rather than a left-adjusted fraction, and Burroughs used sign-magnitude representations for both the exponent and the integer significand.

Some processors had representations for special values:

- CDC processors provided representations for infinity and "indefinite" (what we now call "Not a Number").
- The IBM 7030 Stretch provided representations for infinity, infinitesimal (ε), and OMZ ("Order of Magnitude Zero" with significand = 0).
- The DEC VAX systems provided a "reserved operand" with minus sign and exponent zero that caused an interruption when accessed.

The wide variety of floating-point implementations on past processors led to some of the following oddities that happened on widely used processors:

- Some processors responded to exponent spills by delivering zero, ∞ , MaxReal, or MinReal.
- Some numbers had no inverse.
- A number a had a precisely representable inverse, but $a \times a^{-1} \neq 1$.
- Some processors treated all sufficiently tiny nonzero numbers z as if they were zero during multiplication and division, but not during addition or subtraction.
- $z = y$ but $z-t \neq y-t$ for reasonable values of z, y, and t.
- $1/3 \neq 9/27$.
- $y \times z \neq z \times y$ (for example, the Cray-1 traded commutativity for speed).
- $z \neq 1 \times z \neq 0$.
- $y/z < 0.99$ but $y-z = 0$ (caused by underflow to 0).
- $|z| < 1$, but the processor claimed $|z| \geq 1$.
- The expressions $x < y$ and $(x-y) < 0$ were exactly equivalent on some systems but not on others.
- Some processors had different over/underflow thresholds for multiplication and/or division than for addition and/or subtraction.

These many inconsistencies made it quite difficult (if not impossible) to create reliable numeric software that worked on many systems.[240] This led to the IEEE Floating-Point standard for binary floating-point, adopted in 1975; its format quickly became an industry standard, supplanting most of the formats sketched in Table 398.

[240] Such oddities led someone to propose "Murphy's Law of Floating-Point:" *Anything that can go wrong, does on some computer.*

Exercises

36.8.1.(2) Sketch the formats of the floating-point representations in Table 398 on page 750 for

- Pr1me 550 series single precision
- CDC 6600
- Cray-2 single precision

36.8.2.(2)+ In Table 396 on page 746, the description of Additive Units says that underflow may cause the identity to fail. Give an example in hexadecimal floating-point where this can occur.

36.8.3.(3)+ The list of computational oddities included the possibility that z = y but z−t ≠ y−t. Create "sensible" (non-extreme) short hexadecimal floating-point values that exhibit this behavior.

36.8.4.(2) Search the web to find the representation used by the Burroughs 6700 for short floating-point numbers.

36.8.5.(2)+ Show the character, integer, and the binary, hexadecimal, and decimal floating-point values of these four words: `X'81818181'`, `X'A3A3A3A3'`, `X'F5F5F5F5'`, `X'FEFEFEFE'`.

36.9. Summary

Given the 32-, 64-, and 128-bit floating-point representations, the equivalent decimal precisions that can be faithfully represented are shown in Table 399.

floating-point precision	hexadecimal floating-point	binary floating-point	decimal floating-point
32 bits	6 digits	6 digits	7 digits
64 bits	15 digits	15 digits	16 digits
128 bits	32 digits	33 digits	34 digits

Table 399. Equivalent decimal and floating-point precisions

If precision "equivalent to decimal" is important, decimal floating-point is best.

Some things to keep in mind:

1. Be careful about "optimizing" arithmetic expressions. The possible presence of signed zeros, NaNs, and infinities require care.
2. Predictability is more important than efficiency; getting wrong answers fast helps no one.
3. Respect the parentheses in coded expressions: they often specify a required order of evaluation.
4. Don't accidentally mix operand lengths (except possibly with hexadecimal floating-point, and then only if you're very careful!). A number in a floating-point register may have very different representations; for example, the three binary and decimal representations have different exponent field and significand widths.
5. It is best to do all computations in a single mode: hexadecimal, binary, or decimal, with a single representation for constants and literals.
6. Binary floating-point permits exceptions where none would occur with hexadecimal, such as when shortening an operand or creating inexact and invalid operation exceptions.
7. Inequalities do *not* persist across floating-point operations!
8. Don't compare floating-point values for strict equality; use either an inequality, or a comparison for "equality within an acceptable tolerance".

9. A smaller radix means that relative error grows more slowly, but more exponent bits are needed for a given exponent range. For example, binary requires 2 more exponent bits than hexadecimal for same range.
10. Rounding (as in BFP and DFP arithmetic) compared to truncation (in HFP arithmetic) means that the magnitude of average error tends to be smaller.
11. Don't confuse the “ulp” (for HFP and BFP) with the quantum (for DFP). They might seem to describe the same concept, but an ulp(x) depends on the value of x and may vary by a factor of 2 (BFP) or 16 (HFP), while the magnitude of a quantum depends only on the exponent of the number. An ulp of a DFP number is meaningless, because DFP values aren't normalized.
12. The two signed zeros (in BFP and DFP) are distinguishable arithmetically only by division by zero (producing signed infinities) or using the copy sign instructions. (See Exercise 34.8.3.)

You *must* think differently about floating-point numbers and arithmetic; so why do we rely on floating-point arithmetic?

1. It closely approximates the way most of us do most of our calculations, most of the time.
2. It handles most of the hard work automatically.
3. But, remember that it can be difficult to know with certainty when it *doesn't* work!
4. “The secret of success of floating-point computation lies in the fact that we continue to do arithmetic to p digits of precision even though the accuracy of our intermediate results has degraded so that we can only guarantee that a few digits are significant.”[241]

Terms and Definitions

mantissa
A term sometimes used to describe the significand of a floating-point number. Because it can be confused with the mantissa (fractional part) of a logarithm, avoid its use when describing floating-point representations.

“real” numbers
A powerful abstraction used by mathematicians, in which values have infinite precision and unlimited range.

“realistic” numbers
Numbers used for computation with finite range, precision, and accuracy.

Programming Problems

Problem 36.1.(5) Suppose your processor does not provide instructions to automatically convert between floating-point representations. Write a program that accepts strings of 8 hex digits representing a short binary floating-point value, and converts them to a short hexadecimal floating-point representation using half-up rounding. Display the original and converted values in hexadecimal, and generate appropriate messages if any conversion problems arise.

Suggested test values include: `X'80000000'`, `X'7FFFFFFF'`, `X'7F7FFFFF'`, `X'3DCCCCCD'`, `X'80800000'`, `X'FF800000'`, and `X'007FFFFF'`, but you should create others.

Problem 36.2.(4) Suppose your processor does not provide instructions to automatically convert between floating-point representations. Write a program that accepts strings of 16 hex digits representing a long hexadecimal floating-point value, and converts them to a long binary floating-point representation using rounding to half-even. Display the original and converted values in hexadecimal. If the HFP value is unnormalized, generate a canonical QNaN.

[241] Quoted from *Floating-Point Computation* by Pat H. Sterbenz, Prentice-Hall, 1974.

Suggested test values include: X'80000000 00000000', X'7FFFFFFF FFFFFFFF', X'40199999 9999999A', X'470CCCCC CCCCCCC5', X'80100000 00000000', and X'413243F6 A8885A31', but you should create others.

Problem 36.3.(5) Suppose your processor does not provide instructions to automatically convert between floating-point representations. Write a program that accepts 16 hex digits representing a long binary floating-point value, and converts them to a long hexadecimal floating-point representation using half-up rounding. Display the original and converted values in hexadecimal, and generate appropriate messages if any conversion problems arise.

Suggested test values include: X'80000000 00000000', X'7FFFFFFF FFFFFFFF', X'7FEFFFFF FFFFFFFF', X'3DCCCCCC CCCCCCCD', X'80100000 00000000', and X'FFF00000 00000000', but you should create others.

Problem 36.4.(3) For each floating-point representation, write instructions that will create a table of the values of N Factorial (N!) starting at 1! = 1 in the 8-byte floating-point representation. Determine the maximum representable value without causing overflow or underflow exceptions, and store the value of N for that maximum value as a word integer at HFactMax, BFactMax, and DFactMax for hexadecimal, binary, and decimal floating-point values respectively.

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
PFPF	010A

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
010A	PFPO

Chapter X: Large Programs and Modularization

```
XX         XX
XX         XX
 XX       XX
  XX     XX
   XX   XX
    XXXX
    XXXX
   XX   XX
  XX     XX
 XX       XX
XX         XX
XX         XX
```

The two sections of this chapter discuss program modularization:

- Section 37 introduces the basic concepts of *internal* subroutines, focusing on the fundamental processes of subroutine linkage, argument passing, and status preservation. It then describes the general linkage conventions used on System z operating systems.
- Section 38 describes:
  - Techniques for managing addressability in large programs (and how the powerful LOCTR instruction can help).
  - Separately assembled multi-module applications.
  - How to write *external* subroutines callable from Assembler Language and other programming languages.
  - How separately assembled modules are linked together to form an executable load module or program object, and how they are loaded into memory.
  - How to handle changes in addressing modes.

37. Subroutines and Linkage Conventions

```
 3333333333   777777777777
333333333333  777777777777
33        33  77        77
          33            77
          33           77
       3333            77
       3333           77
          33          77
          33         77
33        33         77
333333333333        77
 3333333333         77
```

Program modularization is a fundamental application development process. We divide a task into smaller pieces that can be written and tested independently. In this and the next section we will examine some general features of subroutines, a basic form of modularization.

Subroutines help you divide a programming problem into smaller and more manageable parts. A calculation needed at many places within a program can be done by including the necessary instructions at each place they are needed, but it is usually more economical to write the needed instructions once, and then execute them as needed.

Of course, if a calculation is needed at only one point in a program, it might seem easiest to insert the instructions there. But we will see that using a subroutine is justifiable because it simplifies program planning, organization, coding, and debugging. Often, a tested subroutine can be used in other programs.

37.1. Basic Concepts

We'll start with three basic concepts:

- *Linkage:* how to pass control to a subroutine and return.
- *Argument passing:* how to provide data needed by the subroutine and access its results.
- *Status preservation:* how to be sure that nothing important is lost, modified, or destroyed in the process.

We call the *main program* the routine that transfers control to the *subroutine*. The former is said to *call* the latter, so the main program is the *calling program* (or *caller*) and the subroutine is the *called program* (or *callee*).

The rest of Section 37.1 uses simple examples to illustrate these three basic concepts; many of them are used in the general linkage conventions discussed in sections 37.2-37.4. For now, we'll consider only 32-bit registers and 24- and 31-bit addressing modes.

37.1.1. Linkage

This problem is relatively simple: we must find a way to transfer control from the caller to the callee and for the callee to then return to the caller. The calling program must know where to transfer control to execute the called program, and the called program must know how and where to return control to the calling program.

To use a trivial (and nonsensical) example, suppose the subroutine starting at an instruction named **ZSet** sets the byte named **Flag** to zero. (For the following examples, assume GR12 has been established as a base register.)

```
         - - -                        Main program calculates something
         J     ZSet                   Then branches to the subroutine (1)
Next     L     2,Noodle               And then continues
         - - -                        ...with something else
         - - -
ZSet     MVI   Flag,0                 Subroutine sets the byte to zero
         J     Next                   And returns to the main program (2)
```

Figure 472. Trivial example of a subroutine (1)

The “processing” performed by this “subroutine” can obviously be done more simply without a subroutine call. The point is that we have (1) transferred control to a subroutine which does something and then (2) returns control to the caller. In its most primitive form, a subroutine can be written as an instruction sequence to which a branch is made, and which returns by branching elsewhere.

Now, suppose we want to call the subroutine starting at **ZSet** from several different places in the main program. We see that the return in Figure 472 always branches to the same instruction (named **Next**) in the calling program. But we may want to do something different before and after calling the subroutine each time. Thus we must solve the second half of the linkage problem: specify where control should be transferred when the subroutine ends.

We can solve this problem by agreeing that when control is transferred to **ZSet**, GR14 will contain a *return address*, the address of the instruction to which the subroutine should branch when it completes. Then we can make two calls to the subroutine at **ZSet** as shown in Figure 473.

```
         - - -                        Calculate something
         LA    14,Next                Set return address in GR14
         J     ZSet                   Branch to subroutine
Next     L     2,Noodle               On return, begin calculating
         - - -                        ...something else
         ST    2,Result               And store it
         LA    14,Ret2                Set new return address in GR14
         J     ZSet                   Branch to subroutine
Ret2     L     3,Poodle               ...and so forth
         - - -
ZSet     MVI   Flag,0                 Subroutine does its work
         BR    14                     And returns to the address in GR14
```

Figure 473. Trivial example of a subroutine (2)

The Branch and Save instructions provide a simpler solution.

37.1.2. The Branch and Save Instructions

The Branch and Save instructions let you execute both the branch to the called program, and set the return address. The four instructions are shown in Table 400 on page 758.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
0D	BASR	RR	Branch and Save (Register)	4D	BAS	RX	Branch and Save
A75	BRAS	RI	Branch Relative and Save	C05	BRASL	RIL	Branch Relative and Save (Long)

Table 400. Branch and Save instructions

We first encountered BASR in Section 10 when we discussed addressability, but there the R_2 digit was assumed to be zero. BAS is an RX-type instruction with the operand format shown in Section 9.5, and BASR is an RR-type instruction with the operand format illustrated in Section 9.2. BRAS and BRASL use relative-immediate addressing, as discussed in Section 20.1.3.

As with other branch-relative instructions, we often use the extended mnemonics JAS for BRAS and JASL for BRASL.

The first step in executing a Branch and Save instruction is determining the *branch address*: for BASR it is the address in the R_2 register (unless R_2 is zero, in which case the instruction doesn't branch) and for the other three instructions the branch address is the Effective Address.

The second step places the Instruction Address (IA) from the PSW into general register R_1, so that R_1 contains the address of the instruction *following* the Branch and Save instruction.[242]

- If the addressing mode is 24 or 31, bit 32 of the R_1 register is set to 0 or 1 accordingly, and bits 0 to 31 are unchanged.
- In 64-bit addressing mode, the 64-bit address of the following (next sequential) instruction is placed in GG R_1.

The third and final step replaces the contents of the IA in the PSW with the branch address, so the next instruction executed will be at the branch address.

Reminder

Remember that the Branch and Save instructions are modal. The address in the R_1 register depends on the current addressing mode, as described in Section 20.2.

We can rewrite Figure 473 on page 757 to use a BAS instruction:

```
         - - -                   Main program labors
         BAS   14,ZSet           Branch to subroutine, set GR14
         L     2,Noodle          On return, labor again
         - - -
         ST    2,Result          Store a new result
         BAS   14,ZSet           Call subroutine again
         L     3,Poodle          ...etc.
         - - -
ZSet     MVI   Flag,0            Subroutine does required work
         BR    14                Return to address in GR14
```

Figure 474. Subroutine linkage using a BAS instruction

This program segment is functionally identical to Figure 473 on page 757 but requires fewer instructions (and symbols) because the BAS instructions provide both the return addresses (formerly generated by LA instructions) and the branches to the subroutine (formerly done by the `J ZSet` instructions).

[242] If the Branch and Save instruction is the subject of an **EX**ecute instruction, R_1 contains the address of the instruction following the execute instruction.

The RX-type BAS instruction requires a base register to resolve the target address; you can also use the relative branch form, JAS, to eliminate the need for one or more base registers for instruction addressing.

A BASR instruction can be used in a similar sequence:

```
         - - -                    Same main program
         LA    15,ZSet            Put subroutine address in GR15
         BASR  14,15              Branch to subroutine
         L     2,Noodle           ...etc
         - - -                    Assume that GR15 is unmodified
         ST    2,Result           ...etc
         BASR  14,15              Call subroutine again
         L     3,Poodle           ...etc
         - - -
ZSet     MVI   FLAG,0             Subroutine
         BR    14                 Returns to caller
```

Figure 475. Subroutine linkage using a BASR instruction

This program segment is identical to that in the previous example, except that the BAS instructions have been replaced by BASR, and GR15 has been preset to contain the "entry point address" of the subroutine.

This convention is widely used for calling subroutines. In Figures 474 and 475, we assumed that the symbol **ZSet** was addressable from the BAS and LA instructions. If the source program is large the entry point of the subroutine may be far away, so you might use a technique like this:

```
         - - -                    Same main program
LZSet    L     15,AdrZSet         Put subroutine address in GR15
         BASR  14,15              Branch to subroutine
         L     2,Noodle           ...etc
         - - -                    Assume that GR15 is unmodified
         ST    2,Result           ...etc
         BASR  14,15              Call subroutine again
         L     3,Poodle           ...etc
         - - -
AdrZSet  DC    A(ZSet)            Addressable from instruction LZSet
         - - -
ZSet     MVI   FLAG,0             Subroutine (quite far from LZset)
         BR    14                 Returns to caller
```

Figure 476. Subroutine linkage using an address constant

Section 38 will explore this theme in more detail.

37.1.3. Argument Passing

Subroutines are rarely this trivial. A subroutine usually operates on data provided to it, and either modifies the data or uses it to compute new values. The data provided to a subroutine are called its *arguments*.[243]

The caller and callee must agree on the method(s) used for passing arguments. We will sketch a few simple techniques that require different amounts of information be known to both routines.

For these examples, suppose we must write a subroutine starting with an instruction named **ShftRt** which is has two arguments: a logical fullword to be shifted and a fullword integer N. The

[243] This terminology comes from the mathematical concept of a function of one or more arguments. Because mathematical functions such as SIN and EXP in higher-level languages are almost always implemented by subroutine calls, we use the same terminology to describe similar situations even when the called routines are not mathematical functions.

subroutine will shift the first argument logically to the right by 2+max(N,0) bit positions: if N is negative shift right two places, and if N is non-negative shift N+2 places.

One simple method places the arguments into agreed-upon general registers. This works only for data which "naturally" fits in registers; a different method would be needed for packed decimal or character data.

Now, let the two arguments for the subroutine at **ShftRt** be placed in registers GR0 and GR1 respectively: the logical argument is in GR0 and the integer N is in GR1. The shifted result is left in GR0 when control is returned to the caller. As above, we assume the return address is in GR14.

```
*        ShftRt Arguments in registers 0 and 1
ShftRt   LTR   1,1                Test value of N
         JNM   ShftOK             Branch if nonnegative
         SR    1,1                Set N to zero if it was minus
ShftOK   SRL   0,2(1)             Shift (2+max(N,0)) bit positions
         BR    14                 Return to caller
```

Figure 477. Simple shift subroutine (1)

While this instruction sequence is straightforward, a possibly undesirable side-effect is created by the subroutine: if the argument N in GR1 is negative, GR1 is set to zero. If the calling program did not need to use the contents of GR1 after the call there is no problem. Otherwise, we might rewrite the routine as in Figure 478:

```
ShftRt   SRL   0,2                Shift logical argument 2 places
         LTR   1,1                Test integer value N
         BNPR  14                 Return to caller if not positive
         SRL   0,0(1)             Otherwise shift N more places
         BR    14                 And return to caller
```

Figure 478. Simple shift subroutine (2)

This is better because the subroutine makes as few changes to the caller's registers as possible.

A second way to pass arguments is to place them in named memory areas. This is sometimes used for operands (such as packed decimal data and character strings) that must be processed in memory. For example, suppose the argument to be shifted is stored in a word named **Logic** and the integer value N is stored in a word named **NN**. Assuming again that the caller expects the shifted result to be returned in GR0, we could write the subroutine as in Figure 479.

```
ShftRt   L     0,Logic            Get argument to be shifted
         L     1,NN               Get number of shifts, N
         LTR   1,1                Check shift count
         JNM   ShftOK             Branch if not negative
         SR    1,1                Set count to zero
ShftOK   SRL   0,2(1)             Shift by required amount
         BR    14                 Return result to caller in GR0
```

Figure 479. Simple shift subroutine with named arguments (3)

In this case we have used the same technique as in Figure 472 on page 757, but no harm is done by the SR instruction that sets GR1 to zero because the value of N stored at **NN** is unchanged. You can mix these techniques; some variations are illustrated in the Exercises.

A more powerful technique is to use argument *addresses.* Then, arguments can be anywhere in memory, and the subroutine is passed the addresses of the arguments. This method can be used for arguments of any type and is preferable as a general scheme; the simpler methods just described are efficient when restricted to particular types of data. First, suppose the address of the logical argument is in GR2, and the address of N is in GR3.

```
*        ShftRt argument addresses in GR2, GR3
ShftRt   L     0,0(0,2)            Get logical argument
         L     1,0(0,3)            Get integer argument
         SRL   0,2                 Shift logical argument 2 places
         LTR   1,1                 Now test integer argument
         BNPR  14                  Return if negative or zero
         SRL   0,0(1)              Shift N places
         BR    14                  And return result in GR0
```

Figure 480. Simple shift subroutine (4) using argument addresses

A key advantage of passing arguments or their addresses in registers is that the subroutine need not know where the arguments are, nor whether they would be addressable within the subroutine.

Now, suppose the argument addresses are also in memory: let the address of the logical argument be in a word named **AdrLogic** and the address of the integer argument N be in a word named **AdrN**. Then we can write the subroutine as in Figure 481:

```
*        ShftRt argument addresses in named memory locations
ShftRt   L     1,AdrLogic          Get first argument address
         L     0,0(0,1)            Get logical argument in GR0
         SRL   0,2                 Shift right 2 bit positions
         L     1,AdrN              Get second argument address
         L     1,0(0,1)            Get integer argument in GR1
         LTR   1,1                 Test sign of N
         BNPR  14                  Return if not positive
         SRL   0,0(1)              Shift remaining N bit positions
         BR    14                  And return
```

Figure 481. Simple shift subroutine (5) with argument addresses in memory

This example is close to the "standard" argument-passing convention used by System z operating systems that we'll discuss in Section 37.2.

In some specialized cases the return address can be used to address the arguments or the argument addresses. This is done by placing them *inline:* the data or data addresses are placed into the instruction sequence following the Branch and Save that links to the subroutine.

To illustrate inline arguments, first suppose the **ShftRt** subroutine is written so that the logical argument is in GR0 on entry and return, but the number of shifts is contained in a halfword *immediately following* the Branch and Save instruction. On entry to the subroutine, R14 contains the address of the halfword integer shift amount, and control must return to the instruction *following* that halfword.

```
         - - -                     Call the subroutine
         L     0,Data1             Get first logical data item
         BAS   14,ShftRt1          Call ShftRt1 subroutine
         DC    H'5'                Shift amount = +5
         ST    0,Result1           Store shifted result
         - - -
         L     0,Data2             Get second data item
         BAS   14,ShftRt1          Call subroutine again
         DC    H'-4'               Shift amount = -4
         ST    0,Result2           Store second result
```

Figure 482. Subroutine call with inline arguments

We can't use the address in R14 for the return address, because we would branch to the halfword constant! The subroutine must account for the 2-byte length of the inline parameter when it returns:

```
*        Subroutine ShftRt1 with inline halfword shift-count argument
ShftRt1  LH    1,0(0,14)          Get halfword shift argument N
         LTR   1,1                Test sign
         JNM   ShftOK             Branch if not negative
         SR    1,1                Clear GR1 to zero
ShftOK   SRL   0,2(1)             Perform required shifts
         B     2(,14) ←           Return, stepping past argument
```

Figure 483. Subroutine returning past inline argument

For a further variation on this inline-argument scheme, suppose the arguments are two fullwords containing the *addresses* of the arguments. The call to the subroutine could then be written:

```
         - - -
         CNOP  0,4                Align to fullword boundary
         BAS   14,ShftRt2         Call ShftRt2 subroutine
         DC    A(Logic,NN)        Argument addresses
         ST    0,Result           Store shifted result
         - - -
Logic    DS    F                  Logical datum to be shifted
NN       DS    F                  Shift amount
Result   DS    F                  Space for result
```

Figure 484. Subroutine call with inline argument addresses

The CNOP statement is needed to guarantee that the 4-byte BAS instruction falls on a fullword boundary, so that no space will be wasted between the BAS and the address constants. Otherwise, a program interruption might be generated by the subroutine. (See Exercise 37.1.6.)

A **ShftRt2** subroutine called with these arguments could be written as follows:

```
ShftRt2  LM    1,2,0(14)          Get argument addresses
         L     0,0(0,1)           Fetch logical datum
         L     1,0(0,2)           ...and shift amount
         SRL   0,2(0)             Shift two places
         LTR   1,1                Now test shift amount
         BNP   8(0,14) ←          If not +, return to caller
         SRL   0,0(1)             Otherwise perform remaining shifts
         B     8(0,14) ←          ...and return
```

In both versions of this subroutine, the RX form of the return branch instruction (rather than RR form) must be used so that the displacement of the BC instruction can contain the length of the arguments to be skipped.

The technique of inline arguments is discouraged today because it can seriously impact program performance, but it was widely used in the past and sometimes appears in instructions generated by operating system macros. It is especially inefficient if the inline arguments vary each time the subroutine is called.

37.1.4. Returned Values

Each of the schemes for passing arguments to the subroutine can also be used for returning computed results to the calling program. (See Exercise 37.1.2.)

When the returned value can be held naturally in a register, we often return it in a register; or, we might store it in a specified place in memory. To illustrate the latter, suppose our general argument-passing scheme above is used with three address arguments. The first two addresses in the address list are the logical word and shift count, and the third address in the address list points to a word where the result is to be stored. Assuming GR1 contains the address of the first of the three argument addresses, we can write the subroutine as follows:

```
*        GR1 has address of argument and result addresses
ShftRt3  LM    2,4,0(1)          Get argument and result addresses
         L     0,0(0,2)          Load logical argument
         L     1,0(0,3)          Load shift amount N
         LTR   1,1               Test sign of shift count
         JNM   ShftOK            Branch if not negative
         SR    1,1               Set shift to zero otherwise
ShftOK   SRL   0,2(1)            Perform the required shifts
         ST    0,0(0,4)          Store result in given location
         BR    14                Return to caller
```

Figure 485. Subroutine with argument address list

The important feature of this example is that GR1 points to a list of argument addresses; we'll see this again in Section 37.3.

37.1.5. Status Preservation

The previous examples assumed that all registers except GR14 were available for use and could be modified by the subroutine. This could conflict with registers used in the calling program. For example, in Figure 485 registers GR0 through GR4 were modified.

Two decisions must be made: what should be preserved, and which routine should do the preserving: caller or callee? Among items that might need to be saved for later use by the caller are the general registers, the floating-point registers, and other things like the Condition Code and Program Mask.

Consider first the problem of determining which routine should do the status preservation. For simplicity, we take this to mean the saving and restoring of the general registers. If the *calling* program saves all the registers before branching to the subroutine, and restores them on return, it might be doing a lot of unnecessary work; a subroutine like **ShftRt** uses very few registers, but the writer of the calling program would typically prepare for the worst and save all the registers.

More critically, whatever registers were used for addressability in the calling program may have been destroyed by the subroutine, which needs only to branch to the instruction at the return address. This could cause serious problems when the caller tries to restore its registers. (See Exercise 37.1.7.)

Thus, it's better for the called routine to save and restore registers. The subroutine can take advantage of its (possibly) economical use of registers by saving and restoring only the ones it modifies.

There are also advantages to not preserving some registers. In Figure 481 on page 761 only GR1 is modified, so there could easily be a convention between caller and callee that GR1 may be used for any necessary purpose without preserving its contents. (Such a register is sometimes called a *scratch* register or a *volatile* register: its contents may “evaporate” across calls.) The extra time and memory needed for status preservation on each subroutine call may be less if we agree to let the subroutine modify and not restore a designated group of registers.

There are many solutions to the problems of status preservation; this example shows a simple technique. Suppose we rewrite Figure 485 so that only GR0 is modified (containing the result), and GR1 through GR3 are restored before returning.

```
*        ShftRt4 subroutine saves and restores GR1-GR3
ShftRt4  STM   1,3,ShftSave       Save GR1 through GR3
         LM    2,3,0(1)           Get argument addresses
         L     0,0(0,2)           Get first argument in GR0
         L     3,0(0,3)           Get second argument in GR3
         LTR   3,3                Test sign of shift amount
         JNM   ShftOK             Branch if non-negative
         SR    3,3                Set shift count to zero
ShftOK   SRL   0,2(3)             Shift by required amount
         LM    1,3,ShftSave       Restore GR1-GR3
         BR    14                 Return to caller
ShftSave DS    3F                 Save area for 3 registers
```

Figure 486. Subroutine saves and restores registers

(See Exercises 37.1.10 and 37.1.11.)

Exercises

37.1.1.(1) Write a version of the **ShftRt** subroutine that receives the logical argument in GR0, and the integer argument N is in a fullword area of memory named NN. The return address is in GR14.

37.1.2.(1) Write a version of the **ShftRt** subroutine that expects its arguments to be in fullword areas of memory named **Logic** and **NN** and which leaves the result in a fullword area of memory named **Result**. The return address is in GR14.

37.1.3.(2)+ What will be found in GR R_1, and what instruction will be fetched next, if the subject instruction of an EX is a Branch and Save?

37.1.4.(3)+ A programmer asked whether or how the actions of the two instructions

```
BASR  2,0     and     BASR  2,2
```

differ; he believed that they are identical except that the latter takes slightly longer. Explain how and why this is incorrect.

37.1.5.(3) Describe the action of the following sequence of instructions:

```
BASR  3,0
- - -                    First block of code
BASR  3,3
- - -                    Second block of code
BASR  3,3
- - -                    Etc.
```

37.1.6.(2)+ In Figure 484 on page 762, what problems might occur if the CNOP instruction is omitted?

37.1.7.(3) Suppose the calling program saves and restores its own registers, and that the called routine returns to its caller using this (nonstandard) instruction sequence:

```
         LR    1,14               Move return address to GR1
         S     1,=F'4095'         Subtract 4095 from ret addr
         LM    2,15,=14F'0'       Have fun with the other regs
         B     4095(,1)           And return to the caller, with
*                                 An odd address in GR1 (HaHa)
```

State the conditions under which the calling program could re-establish its own registers.

37.1.8.(3) The BASR instruction with operands "R_1,R_2", when used for subroutine branching, is functionally identical to the instruction

```
BAS   R1,0(0,R2)
```

Why then is there any use for BASR? Can you think of any reasons why the CPU architects included it in the instruction set? (Apply the same considerations to the instruction pairs BC/BCR and BCT/BCTR.)

37.1.9.(2)+ What will happen at execution time when these instructions are executed?

```
LA    4,XYZ              Address of XYZ in GR4
BASR  5,0
BASR  4,5
```

37.1.10.(2)+ Rewrite the instructions in Figure 486 on page 764 so that only two registers need to be saved and restored.

37.1.11.(2)+ Rewrite the instructions in Figure 486 on page 764 so that only *one* register needs to be saved and restored. Is this solution likely to be more or less efficient than your solution to Exercise 37.1.10?

37.2. A General Linkage Convention

Section 37.1 described basic aspects of *internal* subroutines written as part of your source program. You can use these basic techniques to write internal subroutines using almost any workable conventions. (We'll discuss *external* subroutines in Section 38.)

As your subroutines grow in size and complexity, you will want to use a more standard set of conventions.

- Section 37.3 describes a general scheme for passing arguments.
- Section 37.4 describes status preservation using save areas and register-saving conventions.
- Some additional conventions are described in Section 37.5.

Section 37.4 will summarize the conventions used by most operating systems on System z.

We describe subroutines using several important terms.

- The *entry point* of a subroutine is the first instruction to be executed when it receives control from the caller.
- The *return address* is set by the caller to the address of the first instruction to receive control when the subroutine terminates its execution.
- Values passed to a subroutine are *arguments* and variables using those values in the subroutine are *parameters*.

  This example may help: suppose we need a subroutine **Add2** that adds two integers and stores their sum. We might define Add2(X,Y,Z) to mean “Add the values of X and Y and store their sum at Z”, even though we don't know the values associated with X and Y. We might think of the subroutine doing something like this:

  ```
  L     0,X                Get first value from caller
  A     0,Y                Add second value from caller
  ST    0,Z                Store sum where caller specifies
  ```

  If we want the **Add2** subroutine to add the integers at A and B and store their sum at C, we could write something like “CALL Add2(A,B,C)”. In this case, A, B, and C are the *arguments* of the call to Add2, and X, Y, and Z are the *parameters* of Add2. We *associate* arguments and parameters in order from left to right: argument A is associated with parameter X, B with Y, and C with Z. Another invocation like “CALL Add2(D,E,F)” will associate D with X, E with Y, and F with Z.

  Thus, *arguments* have values supplied by the caller, while *parameters* are place-holders in the called subroutine. Parameters let the called routine use its own names to refer to the caller's arguments (whose names may not be known to the callee); this is especially true for separately assembled routines.

37.3. Argument Passing

The previous examples have illustrated many ways to pass arguments. One further refinement provides a very general method. Suppose, as in Figure 486 on page 764, that the addresses of the arguments are in successive words in memory, and when the subroutine is called register GR1 contains the address of the first address; this is illustrated in Figure 487, where we use 32-bit addresses. (We'll discuss 64-bit addresses shortly.)

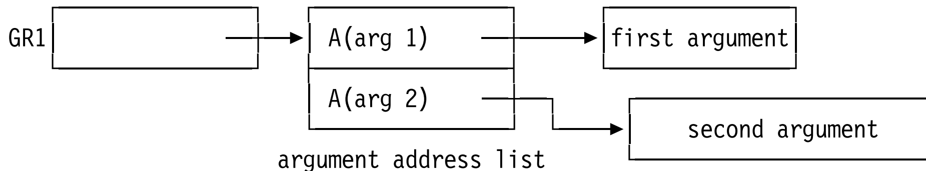


Figure 487. General argument-passing scheme

The argument addresses in the successive fullwords in memory form the *argument address list.*

Figure 488 illustrates a call to a **ShftRt5** subroutine; the first instruction is **L**oad **A**ddress, not **L**oad, so GR1 contains the *address* of the argument address list.

```
         LA    1,ArgList          GR1 points to address list
         BAS   14,ShftRt5         Call ShftRt5 subroutine
         ST    0,Result           Store shifted result
         - - -
Logic    DS    F                  Argument to be shifted
ArgList  DC    A(Logic,NN)        Argument address list
Result   DS    F                  Space for result
NN       DS    F                  Number of shifts
```

Figure 488. Subroutine call using an argument address list

The arguments (Logic and NN) could have had any names and could be anywhere in the program; only the two address constants containing their addresses need be contiguous (and in this example, addressable). A **ShftRt5** subroutine using this argument-passing convention is shown in Figure 489.

```
*        Argument address list pointer in GR1
ShftRt5  LM    2,3,0(1)           Get argument addresses in GR2, GR3
         L     0,0(0,2)           Get 1st (logical) argument in GR0
         L     1,0(0,3)           Get 2nd (integer) argument in GR1
         SRL   0,2                Shift 2 places
         LTR   1,1                Test second argument
         BNPR  14                 Return if it's not positive
         SRL   0,0(1)             Otherwise shift N places
         BR    14                 And return
```

Figure 489. Subroutine called with an argument address list

With the exception of the first instruction, this example is essentially the same as in Figure 480 on page 761.

If we want to call this subroutine with other arguments, we can write instructions to build the argument address list, as illustrated in Figure 490 on page 767.

```
         ST    7,LogicTmp          Store a different logical variable
         ST    4,NTemp             Store a different shift count
         LA    0,LogicTmp          c(GR0) = A(LogicTmp)
         LA    1,NTemp             c(GR1) = A(NTemp)
         STM   0,1,ArgList         Store argument addresses in list
         LA    1,ArgList           c(GR1) = A(argument address list)
         JAS   14,ShftRt5          Call the ShftRt5 subroutine
         - - -
ArgList  DS    2A                  Space for 2 argument addresses
NTemp    DS    F                   Space for a shift count
LogicTmp DS    F                   Space for a logical variable
```

Figure 490. Constructing an argument address list

37.3.1. Variable-Length Argument Lists

Sometimes a subroutine may accept a variable number of arguments. Thus, we need a way for a subroutine to determine the number of arguments passed to it. This is done by setting the leftmost bit of each 32-bit argument address to zero, except for the last argument address where we set its leftmost bit to 1. For example, to call a subroutine with two and then with three arguments, we could define the argument address lists as

```
ArgList1 DC    A(ArgA1,ArgA2+X'80000000')        Two arguments
ArgList2 DC    A(ArgB1,ArgB2,ArgB3+X'80000000')  Three arguments
```

Figure 491. Two variable-length argument lists

and the subroutine can determine the number of arguments passed to it. To illustrate, suppose a subroutine named **AddHW** is called with these two argument address lists. The routine is to return the sum of the halfword integer arguments in GR0. We can call the subroutine with instructions like these:

```
         LA    1,ArgPtrs           GR1 points to argument list
         JAS   14,AddHW
         - - -
HiBit    Equ   X'80000000'         High-order bit for 32-bit address
ArgPtrs  DC    A(Arg1,Val4,Whenever,Track+HiBit)  Arg address list
Track    DC    H'13030'            A halfword value
Val4     DC    H'-7294'            Another
Whenever DC    H'2016'             Another halfword value
Arg1     DC    H'12345'            Still another halfword value
```

Figure 492. Calling a subroutine with a variable-length argument list

This example shows that the arguments can be "anywhere" in the calling program, not necessarily in any order. The **AddHW** subroutine could be written as in Figure 493:

```
AddHW    SR    0,0                 Clear GR0 for sum
AddLoop  L     2,0(0,1)            Pick up an argument address
         AH    0,0(0,2)            Add the argument to the sum
         LA    1,4(0,1)            Increment address list pointer
         LTR   2,2                 Test if we just added the last
         JNM   AddLoop             Branch if not done
         BR    14                  Otherwise return sum to caller
```

Figure 493. Subroutine called with a variable-length argument list

The LTR instruction checks the high-order bit of the address of the just-added argument to test whether it was the last in the list.

In 24- and 31-bit addressing mode, where addresses can be held in a 32-bit register and word, a variable-length argument list looks like this:

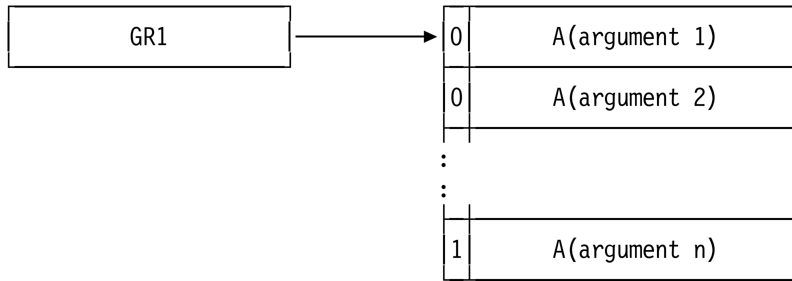


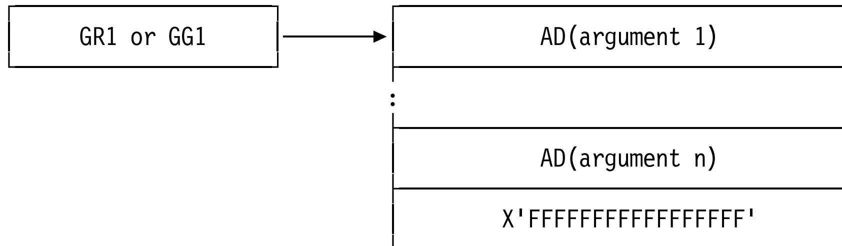
Figure 494. Sketch of a variable-length argument list

This is the standard argument address list for 32-bit addresses: the high-order bit of the address of the last argument is set to 1.

37.3.2. Argument Lists with 64-Bit Addresses

At the time of this writing, no conventions have been established for variable-length lists of 64-bit argument addresses, or for setting a return flag. Some possible methods include the following; all require agreement between caller and callee; each method assumes GR1 (or GG1) contains the address of the list.

1. After the last argument address, add a doubleword of all one-bits. (This is sometimes called a "fence".)



2. After the last argument address, add a doubleword of all zero-bits.
3. Immediately precede the list with a halfword integer containing the number of argument addresses. (This requires correct alignment of the halfword; see Exercise 37.3.5.)

```
         ORG   *+2,8,-2           Align to halfword before doubleword
Arglist  DC    Y(NArgs)           Number of argument addresses
         DC    AD(Argument_1)     Address of 1st argument
         - - -
         DC    AD(Argument_n)     Address of nth argument
NArgs    Equ   (*-(Arglist+2))/8  Number of argument addresses
```

Figure 495. Sample 64-bit argument list addresses

4. The first doubleword in the argument list contains the number of argument addresses that follow it:

```
Arglist  DC    AD(NArgs)          Number of argument addresses
         DC    AD(Argument_1)     Address of 1st argument
         - - -
         DC    AD(Argument_n)     Address of nth argument
NArgs    Equ   (*-Arglist)/8-1    Number of argument addresses
```

Exercises

37.3.1.(2) Write a version of the **ShftRt** subroutine that uses an argument address list like that in Figure 487 on page 766. The first and second addresses point to the word to be shifted and the shift count. The third address in the list is the address of the fullword area where the result

is to be stored. The return address is in R14. Restore all registers to the contents they had on entry to the subroutine.

37.3.2.(3)+ Write a subroutine starting at an instruction named **AMax** that has two arguments, passed according to the general scheme illustrated in Figure 487 on page 766. The first argument is the lowest addressed element of an array of word integers, and the second argument is a positive word integer containing the number of elements in the array. The subroutine should return to the address in GR14 with the largest element of the array in GR0. The subroutine should save and restore the contents of any registers modified by the subroutine.

37.3.3.(3) Write a subroutine starting at an instruction named **CMax** like that in Exercise 37.3.2, except that the array elements are addresses pointing to character strings prefixed with a halfword length field containing the number of bytes in the following string. Compare the strings, and return in GR1 the address of the string comparing higher than any of the others. Assume that shorter strings are padded with blanks. (Hint: the CLCL instruction may help.)

37.3.4.(3)+ Write a subroutine to count the number of 1-bits in a string of bytes. The subroutine entry point should be named **NBITS** and a typical standard call from a high-level language would look like

```
CALL NBITS( String, StrgLen, NCount )
```

The first argument is the address of the first byte of the string, the second argument is a word integer containing the number of bytes in the string (not including the length of this word!), and the third argument is where the count of 1-bits should be stored. If the second argument is not strictly positive, set the bit count to zero.

37.3.5.(2)+ In Figure 495 on page 768, explain why the first operand of the ORG instruction is `*+2` instead of simply `*`.

37.3.6.(2)+ Criticize the following schemes for passing a variable-length list of arguments to a subroutine, using 32-bit argument addresses, and compare them to the standard convention.

1. The number of arguments is contained in the leftmost byte of R1 on entry to the subroutine.
2. The end of the argument list is indicated by the presence of a fullword zero following the last valid argument address.
3. The first fullword in the argument address list holds the number of argument addresses in the rest of the list.

37.3.7.(2) The IBM Model 026 card punch produces holes in a card according the the old BCD character representation. The important differences between it and the EBCDIC representation are that the BCD characters `'=+()` punched by an 026 produce the *same* hole configurations as the characters `@#&><` in the EBCDIC representation produced by a Model 029 card punch.

Write a subroutine named **CONVT** that will convert all occurrences of the characters `@#&><` in a character string to the characters `'=+()` respectively. That is, replace each `@` by `'`, each `#` by `=`, each `&` by `+`, and so forth. The subroutine should use standard linkage, parameter passing, and status-preservation conventions. There are two parameters: the string of characters, and the length of the string. (Table 170 on page 430 may help.)

37.3.8.(2)+ Write a new version of the **ShftRt** subroutine, now named **ShftRt64**, that is called in 64-bit addressing mode, using an argument address list like the one shown in the first example in Section 37.3.2 (that is, with 64-bit argument addresses terminated with a doubleword of all 1-bits). Assume that the arguments themselves are 32-bit integers.

37.4. Save Areas

We'll start with conventions used for 32-bit registers.

If a register save area is used within a program, we must necessarily modify some part of the program itself. While this is not a problem for many applications, there are times when we want to write *re-enterable code* that can be shared by many simultaneously executing programs. This requires different techniques for allocating save areas.

For now, we'll consider programs that contain internal save areas. Such programs are not re-entrant; we'll see the changes needed to support re-entrant programs in Chapter XII.

By convention, the caller provides a “standard” 18-word save area, and its address is passed to the callee in GR13. The caller's general registers are stored starting at offset +12 in the order GR14, GR15, GR0, GR1, GR2, ..., GR12. The easiest way to save the registers is to execute the instruction

```
         STM   14,12,12(13)      Save GR14-GR12 in caller's save area
```

before the called program modifies any of them. This is often one of the first instructions executed by a called program.

Table 401 shows the contents of a standard “Format-0” 18-word save area. (Note that words are numbered starting at zero.)

Word	Offset	Offset	How this word is used
0	0	X'0'	Special Purpose Data; not to be modified
1	4	X'4'	Address of caller's (A's) save area; stored here by the called routine (B) owning this save area (back chain)
2	8	X'8'	Address of the save area in the subroutine most recently called by this routine; put here by that subroutine (C) (forward chain)
3	12	X'C'	c(GR14) (return address to caller)
4	16	X'10'	c(GR15) (entry point address of this routine)
5	20	X'14'	c(GR0)
6	24	X'18'	c(GR1)
7	28	X'1C'	c(GR2)
...	...	...	...
17	68	X'44'	c(GR12)

Table 401. Standard (Format-0) Save Area

Assuming that our current routine B was called by routine A, and that B will call routine C, Figure 496 on page 771 shows a sketch of a standard save area that we provide:

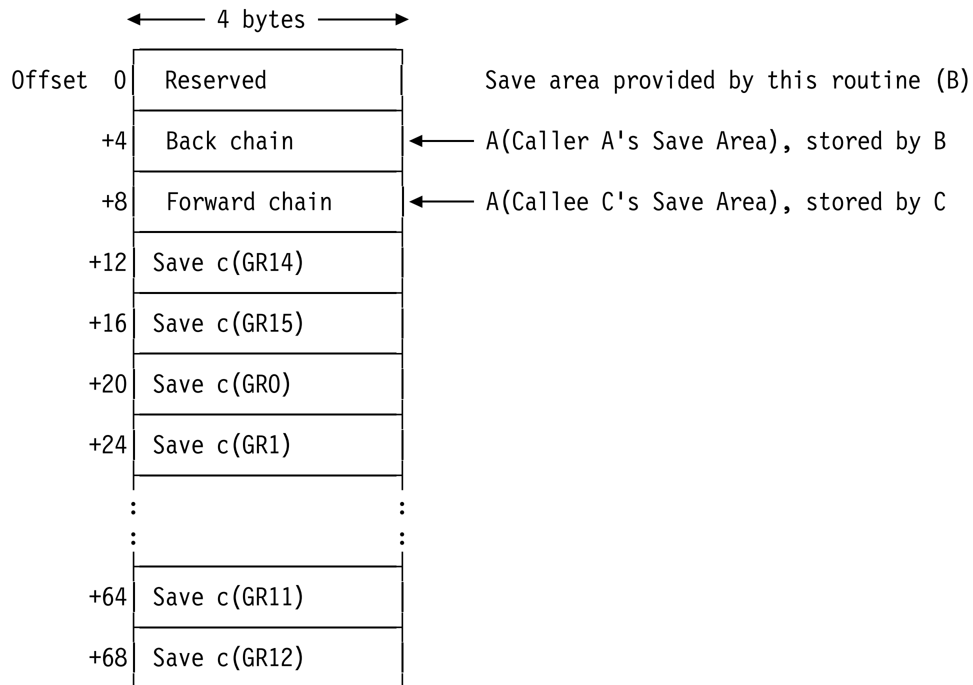


Figure 496. Standard save area layout

The first word of the save area is used for special purposes by some high-level languages, and should not be modified in any way.[244] The second and third words are used to chain (link) the save areas in a doubly-linked list.

Suppose the save area is in routine B, that routine B was called from routine A, and that B will call routine C, as in Figure 497.

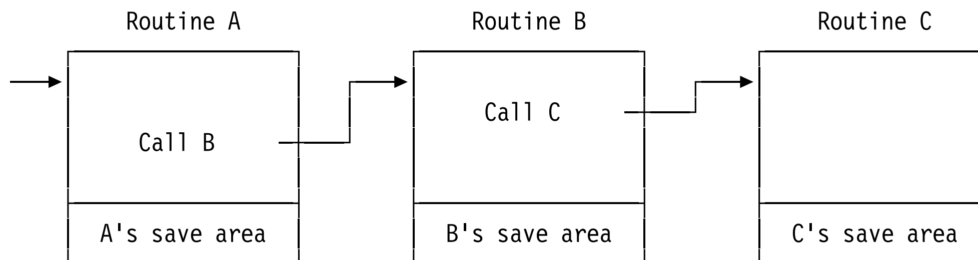


Figure 497. Sample subroutine calling sequence

To show how save areas are used and how they are *chained*, we suppose routine B is called by routine A. Figure 498 on page 772 shows instructions used at the entry point of B; these are typical of many subroutines.

[244] In your Assembler Language routine it may be prudent to zero the first word of the save area you provide to routines *you* call. But don't touch the first word of any other routine's save area if it was created by a high-level language.

```
B      STM   14,12,12(13)       Save registers in A's area
       BASR  12,0               Establish local base for B
       Using *,12               Addressability for B
       ST    13,BSave+4         Store address of the caller's
*                               ... area at word 1 of our area
*                               ... (back chain from B to A)
       LR    2,13               Copy GR13 temporarily to GR2
       LA    13,BSave           Establish address of our area
*                               ... in GR13 for calls to C
       ST    13,8(0,2)          And store the address of our
*                               ... save area in A's save area
*                               ... (forward chain from A to B)
       - - -
BSave  DC    18F'0'             Save area in routine B
```

Figure 498. Save area chaining instructions

These instructions put the address of B's save area into R13, so that routine B can call lower-level routines such as C. After a sequence of subroutine calls, A to B to C to ..., the chained save areas would then appear as in Figure 499.

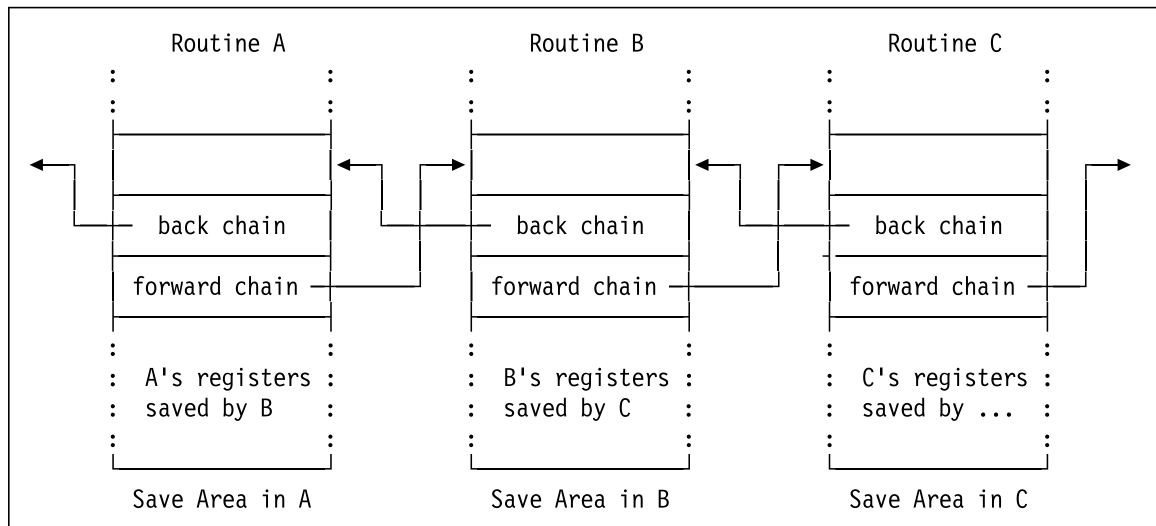


Figure 499. Chained save areas

To return control to the calling program, we restore the registers and branch to the return address. For a routine having its own save area (as for B, in Figure 498), the address of the save area in the *calling* program A must be retrieved before A's registers can be reloaded. Because the address of B's save area is in GR13, we use the typical subroutine return instructions shown in Figure 500.

```
L     13,4(,13)          Retrieve address of caller's area
LM    14,12,12(13)       Restore all registers
BR    14                 Return to caller
```

Figure 500. Reloading registers and returning to a caller

It is not necessary to save and then restore *all* the general registers, so long as the others are not modified in any way.

37.4.1. Extended Save Area Conventions (*)

When your programs are executed in an environment using 64-bit registers, the save area conventions can be more complex. There are three situations we will consider:

1. Your calling and called programs use only 32-bit registers.

   You don't need to do anything different; continue to use the standard (Format-0) save areas shown in Table 401 on page 770.

2. Your calling program has provided a 144-byte save area, or your program and its caller are executing in 64-bit addressing mode.

   Use the "Format-4" save area described in Section 37.4.2, Table 402.

3. Your calling program uses only 32-bit registers (and provides a standard 72-byte, 18-word Format-0 save area), but your (called) program uses 64-bit registers. We'll assume that both calling and called programs execute in 24- or 31-bit addressing mode.

   Use the "Format-5" save area described in Section 37.4.3, Table 403 on page 775.

As before, we assume for sake of simplicity that save areas are internal to the programs.

37.4.2. Format-4 Save Area Conventions for 64-bit Registers (*)

We can't use a standard 18-word save area because forward and back chains are now 64-bit addresses. Instead, we use the 144-byte "Format-4" save area shown in Table 402. The chain-address fields are now at the end of a Format-4 save area. (Note that words are numbered starting at zero.)

The second word of a Format-4 save area contains the identifying characters F4SA. This is used by diagnostic tools to know how to display the contents of the save area.

Word	Offset	Offset	How this field is used
0	0	X'0'	Special Purpose Data; not to be modified
1	4	X'4'	C'F4SA' (Used by diagnostic tools to properly display register contents and locate chain addresses.)
2-3	8	X'8'	c(GG14) (return address in the caller)
4-5	16	X'10'	c(GG15) (entry point address in the called routine)
6-7	24	X'18'	c(GG0)
8-9	32	X'20'	c(GG1)
10-11	40	X'28'	c(GR2)
...	...	...	...
30-31	120	X'78'	c(GG12)
32-33	128	X'80'	Address of the save area in the subroutine that called this routine (back chain)
34-35	136	X'88'	Address of the save area in the subroutine most recently called by this routine; put here by that subroutine (forward chain)

Table 402. Standard Format-4 save area

A representation of a Format-4 save area is shown in Figure 501 on page 774.

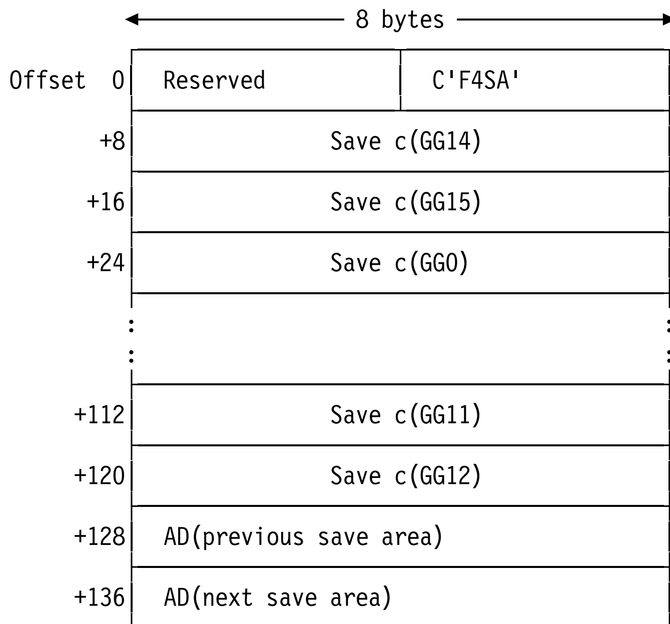


Figure 501. Format-4 save area layout

To show how to use a Format-4 save area in a program executing in 64-bit addressing mode that was called by a program providing a Format-4 save area, you might use instructions like those in Figure 502:

```
CSECT64  CSect ,
CSECT64  AMode 64                Addressing mode 64
CSECT64  RMode Any               Residence anywhere below the 2GB “bar”
         STMG  14,12,8(13)       Save 64-bit registers in caller's area
         LARL  12,CSECT64        Set local base register
         Using CSECT64,12        Establish addressability
         LA    11,MySave         Point to local save area
         STG   11,136(,13)       Save forward link in caller's area
         STG   13,MySave+128     Save backward link in local area
         LGR   13,11             Set GG13 to point to local area
         - - -                   Do good things
Return   LG    13,MySave+128     Retrieve caller's are address
         LMG   14,12,8(13)       Restore registers
         BR    14                Return
MySave   DC    F'0',C'F4SA',17D'0'     Format-4 save area
```

Figure 502. Example of using a Format-4 save area

37.4.3. Format-5 Save Area Conventions for 32- and 64-bit Registers (*)

In this case, we assume that the caller *and* your routine are executing in 24- or 31-bit addressing mode, and that your (the called) routine wants to use all 64 bits of some of the general registers. Your caller has provided a standard 18-word save area, but your routine needs to save the high-order halves of the general registers because the calling program might also be using all 64 bits of the registers.

In this case, you need a 208-byte Format-5 save area for *your* routine, as described in Table 403 on page 775 and illustrated in Figure 503 on page 775.

Word	Offset	Offset	How this field is used
0	0	X'0'	Special Purpose Data; not to be modified
1	4	X'4'	C'F5SA' (Used by diagnostic tools to properly display register contents and determine chaining links.)
2-3	8	X'8'	c(GG14) (return address in A)
4-5	16	X'10'	c(GG15) (entry point address in B)
6-7	24	X'18'	c(GG0)
8-9	32	X'20'	c(GG1)
10-11	40	X'28'	c(GR2)
...	...	...	...
30-31	120	X'78'	c(GG12)
32-33	128	X'80'	Address of the save area in the subroutine (A) that called this routine (B) (back chain)
34-35	136	X'88'	Address of the save area in the subroutine most recently called by B; put here by that subroutine (C) (forward chain)
36-51	144	X'90'	High halves of c(GG0)-c(GG15)

Table 403. Standard Format-5 save area

Figure 503 shows the layout of a Format-5 save area.

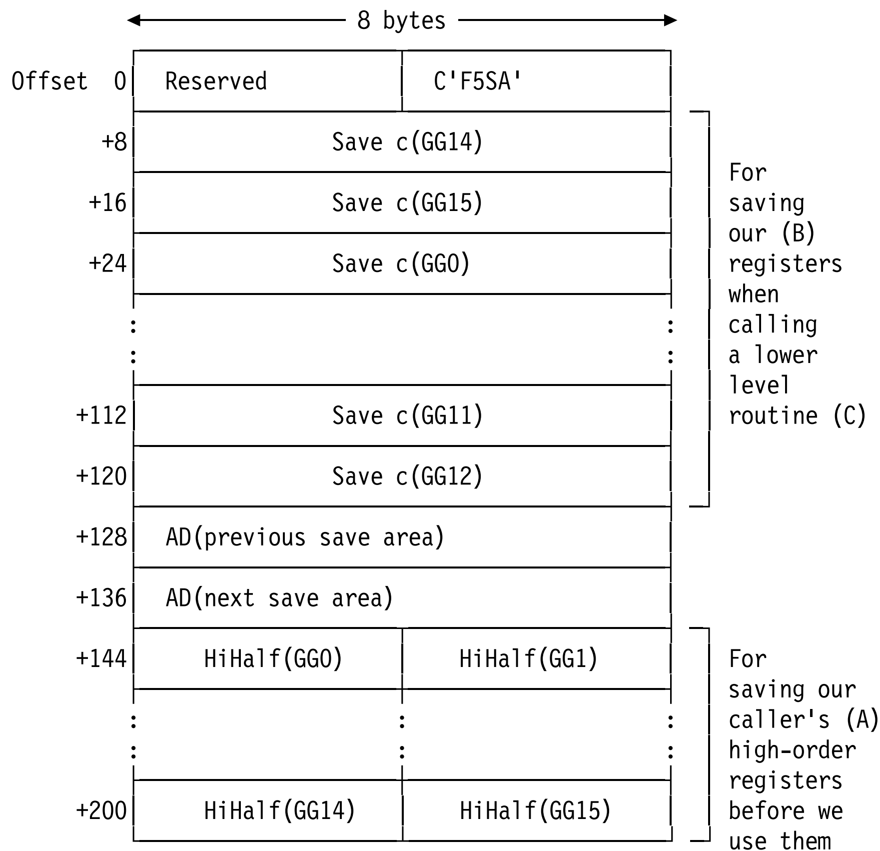


Figure 503. Format-5 save area layout

First, you save the caller's 32-bit registers in *his* save area, as in Figure 498 on page 772. Then, you save the high-order halves of all 16 general registers in *your* Format-5 save area in the 16 words starting at offset +144, as illustrated in Figure 504 on page 776.

```
BProg    CSect ,
BProg    RMode Any               Residence below the 2GB “bar”
BProg    AMode Any               24- or 31-bit addressing mode
         Using *,15
         STM   14,12,12(13)      Save low-order halves of GG14-GG12
         STMH  0,15,BSaveHiH     Save high-order halves of GG0-GG15
         STG   13,BSaveBCh       Save 64-bit back chain address
         LA    1,BSave5          31-bit addr of our Format-5 save area
         ST    1,8(,13)          Store forward chain in caller's area
         LGR   13,1              Point GG13 to our save area
         - - -
BSave5   DS    0D                208-byte Format-5 save area
         DS    F                 Reserved
         DC    C'F5SA'           Format identifier
         DS    15D               Save area for GG14-GG12 (if we call C)
BSaveBCh DS    D                 Back chain address
BSaveFCh DS    D                 Forward chain address
BSaveHiH DS    8D                Save high halves of GG0-GG15
```

Figure 504. Saving registers using a Format-5 save area

After executing these instructions your program can use the 64-bit general registers. When it's time to return to the caller, the high-order halves of the registers must be restored, as in Figure 505.

```
         LG    13,BSaveBCh       Get address of caller's save area
         LMH   0,15,BSaveHiH     Reload high-order halves
         LM    14,12,12(13)      Reload low-order halves
         BR    14                Return to caller
```

Figure 505. Return from a routine using a Format-5 save area

To summarize the three main types of save area:

Format-0 The traditional 72-byte save area that saves only the low-order 32 bits of the general registers.

Format-4 C'F4SA' in the second word indicates a 144-byte save area for all 64 bits of the general registers.

Format-5 C'F5SA' in the second word indicates a 208-byte save area used when a program executing in 24- or 31-bit addressing mode calls a program also executing in 24- or 31-bit addressing mode, *and* wants to use all 64 bits of the general registers. The first 144 bytes are the same as in the Format-4 save area. The low-order 32 bits of the caller's registers are saved in his Format-0 save area, and the high-order 32 bits of the caller's registers are saved in the 64 bytes starting at offset 144.

If the caller or callee is executing in 64-bit addressing mode, many more instructions may be needed to save and restore the status of the caller.

Note

There are other forms of save area you can use if your application involves Access Registers (not discussed here).

Exercises

37.4.1.(2)+ The second word of a Format-4 or Format-5 save area contains the characters F4SA or F5SA. Why can this not be confused with the back-chain address of a standard save area?

37.4.2.(2) In Figure 505, the LMH instruction changes the high half of GG14. Why won't this affect the return address?

37.4.3.(2)+ Give the lengths in hexadecimal of Format-0, Format-4, and Format-5 save areas.

37.5. Additional Conventions (*)

Some other conventions can be used in subroutine calls: entry point identifiers, calling point identifiers, return flags, and return codes. Other than return codes, they are used largely for program debugging and error diagnostics: if your program terminates "abnormally" the Supervisor's diagnostic programs can check for these three items and display them as part of memory dumps and other "post-mortem" information. Also, entry point identifiers and calling point identifiers can be used to provide execution-time flow tracing information.

37.5.1. Entry Point Identifiers (*)

An entry point identifier is a string of characters describing the called routine's entry point in some way; a typical identification is the name of the entry point. The entry point identifier (sometimes abbreviated EPID) is constructed as follows:

1. The first instruction at the actual entry point is an unconditional branch to the STM instruction that will save the registers.
2. The next byte contains a count of the number of characters in the following character string.
3. The identifier string of 1 to 255 characters follows immediately after the count byte.
4. The following instruction is the halfword-aligned STM instruction that saves the registers in the caller's save area.

Figure 506 shows how an entry point identifier is created:

```
         Using *,15                Use caller's preset GR15
Sample   B     Saver               Branch to STM to save regs
         DC    AL1(L'EPID)         Length of identifier
EPID     DC    C'Sample'           Entry point identifying string
Saver    STM   14,2,12(13)         Save registers
         - - -                     etc...
```

Figure 506. Example of an entry point identifier

In this case the ID string is the name of the entry point instruction, but it can be any useful information:

```
         Using *,15                Use caller's preset GR15
Sample   B     Saver               Branch to STM to save regs
         DC    AL1(L'EPID)         Length of identifier
EPID     DC    C'Sample routine''s EPID created 30 Feb 2035, 14:45 15 Mar 2016, 12:41'
Saver    STM   14,2,12(13)         Save registers
         - - -                     etc...
```

To save you having to worry about the details of identifying an entry point and storing registers, the "Library of Macro Instructions" available to the Assembler (see Figure 23 on page 72) will usually contain a SAVE macro instruction. You use it in one of these three forms:

```
name     SAVE  (R1,R2)
name     SAVE  (R1,R2),,character-string
name     SAVE  (R1,R2),,*
```

where the name entry is optional.

The first form generates only the a STM instruction which saves registers R_1 through R_2 in the caller's save area. The second generates the "character-string" as an entry point identifier, and then the STM instruction. The third form generates an entry point identifier consisting either of the name-field symbol name if it's present or the name of the control section in which the SAVE appears, and then generates the STM instruction.

Thus, we could have written the last four statements of Figure 506 as the single statement

```
Sample   SAVE  (14,2),,*         Identify entry, save registers
```

and the generated instructions would look like these:

```
Sample   B     12(0,15)                        BRANCH AROUND ID
         DC    AL1(6)
         DC    CL6'Sample'                     IDENTIFIER
         STM   14,2,12(13)                     SAVE REGISTERS
```

For more about using macro instructions like SAVE, see the IBM publications for your operating system environment.

37.5.2. Calling Point Identifiers (*)

A calling point identifier is a NOP instruction (described in Section 15.4) following the BAS or BASR. It contains a halfword integer like an “identification number” or statement number. For example, we can call a subroutine named **Dummy** with the two variable-length argument lists in Figure 491 on page 767:

```
         L     15,=A(Dummy)      'Dummy' is a separate subroutine
         LA    1,List1           Point to first argument address list
         BASR  14,15             Link to subroutine
         NOP   1                 Calling point ID = 1
         L     15,=A(Dummy)      As above (making sure GR15 is correct)
         LA    1,List2           Point to second arg address list
         BASR  14,15             Link to subroutine
         NOP   2                 Calling point ID = 2
```

Figure 507. Example of two calling point identifiers

When the called routine returns to the instruction whose address was passed to it in GR14, control will immediately continue to the instruction after the NOP.[245] The logic of your program is not affected by the presence of a calling point identifier.

37.5.3. Save Area Return Flags (*)

A save area return flag is helpful as a diagnostic device for programs that might terminate unexpectedly. Suppose there isn't enough information in the registers for you to determine which routine was most recently in control. You can examine the chain of save areas in a memory display to tell which routines have called one another, but you might not be able to tell *which* routines have returned to their callers. This added bit of information is provided by a return flag.

After the caller's registers have been reloaded and just before branching to the instruction whose address is in R14, the callee executes the instruction

```
         OI    15(13),X'01'
```

This sets the low-order bit of the fourth word of the caller's save area to 1. Since the caller's GR14 (saved at offset +12) has been reloaded before the OI is executed, the contents of GR14 on return is the same as when the call was made. We can revise the example in Figure 500 on page 772 to set the return flag:

```
         L     13,BSave+4        Retrieve address of caller's area
         LM    14,12,12(13)      Restore all registers
         OI    15(13),X'01'      Set return flag in save area
         BR    14                Return to caller
```

Figure 508. Setting a return flag

[245] With certain options, the High Level Assembler will view the NOP instruction like any other branch, and check whether the Effective Address appears to reference the lowest 4K bytes of memory without a base register. In such cases, you might write `NOP 1(,0)`, or specify other Assembler options.

If you wonder why the low-order bit of the byte at offset +15 in the save area might not already be 1: if your program was not invoked in 64-bit addressing mode with the BASSM or BSM instructions, it's the address of an instruction to which control is returned, so it must be an even address. If your program was invoked in 64-bit mode with BASSM or BSM instructions, then this bit is already on and cannot be used to convey other information.

Be careful!

If your subroutine can execute in any addressing mode (including 64-bit mode) and is called by a caller executing in 24- or 31-bit mode, don't use this technique to set a return flag. If your program was invoked in 64-bit mode, this bit was already set to 1 and can't be used to convey return information.

We'll see the reasons for this when we discuss mode-changing instructions in Section 38.

37.5.4. Return Codes (*)

Return codes are used often when you communicate with Operating System services. They are usually small nonnegative integers, returned to the caller in GR15. Normally, the calling routine won't expect the contents of R15 to be restored, so the called routine can use GR15 to pass a return code to the caller. This could be any value, but by convention only multiples of 4 such as 0, 4, 8, and so forth are used as return codes.

For example, suppose a subroutine does a calculation and returns the result in GR0, with GR15 set to zero. If the subroutine has been passed an invalid argument (so that it can't do the calculation correctly), it can set GR15 to +4 to indicate the error condition and let the caller analyze and maybe correct the situation.

The instructions in Figure 509 show how the subroutine could set the return code for a “good” and a “bad” return.

```
*        - - -                  Valid calculated result is in GR0
GoodRet  L     13,Save+4        Retrieve pointer to caller's area
         L     14,12(0,13)      Restore caller's GR14
         LM    1,12,24(13)      Restore GR1 through GR12
         SR    15,15            Set return code to zero
         OI    15(13),X'01'     Set return flag
         BR    14               Return to caller
*        - - -                  Invalid argument, no result in GR0
BadRet   L     13,Save+4        Retrieve addr of caller's save area
         LM    14,12,12(13)     Restore all the registers
         LA    15,4             Set return code to +4
         OI    15(13),X'01'     Set return flag
         BR    14               Return to caller
```

Figure 509. Setting a return code in register 15

Another way to set a return code that might have been given different values in the called program:

```
         L     13,4(,13)        Retrieve caller's save area address
         L     14,12(,13)       Restore return address
         L     15,Retcode       Put return code in GR15
         LM    0,12,20(13)      Restore remaining registers
         BR    14               Return to caller
 or
         L     13,4(,13)        Retrieve caller's save area address
         MVC   16(4,13),Retcode Put return code in GR15 slot
         LM    14,12,12(13)     Restore registers
         BR    14               Return to caller
```

Figure 510 on page 780 shows how the calling program can test the return code to validate the result:

```
         L     15,=A(Sub)          Entry point address in GR15
         LA    1,ArgLst            Argument address list
         BASR  14,15               Link to subroutine
         LTR   15,15               Test return code in GR15
         JNZ   Error               If nonzero, error indicated
         ST    0,Result            Store valid result
         - - -
ArgLst   DC    A(DataItem)         Address of data for 'Sub'
```

Figure 510. Testing a return code returned in register 15

When the return code can take on more values, it can be used as an index into a list of branch instructions.

```
         - - -                     Set up registers for linkage
         BASR  14,15               Link to subroutine as usual
         CHI   15,MaxRC            Check for return code too large
         JH    BadRC               Go do something if so
Br2List  B     *+4(15)             Indexed branch into list
         J     Ret000              Return code = 0
         J     Ret004              Return code = 4
         J     Ret008              Return code = 8
         J     Ret012              Return code = 12
         - - -                     etc...
```

Figure 511. Using a return code as a branch index

If you are using relative branch instructions to minimize (or eliminate) the need for base registers to address your instructions, you can't use the RX-type B instruction named **Br2List** in Figure 511. Instead, use instructions like those in Figure 512:

```
         - - -                     Set up registers for linkage
         BASR  14,15               Link to subroutine as usual
         LARL  14,JList            Address of branch list
         AR    14,15               Add return code to list address
         BR    14                  One-hop branch to correct routine
JList    J     Ret000              Return code = 0
         J     Ret004              Return code = 4
         J     Ret008              Return code = 8
         J     Ret012              Return code = 12
         - - -                     etc...
```

Figure 512. Using a return code as a branch index with relative branch instructions

It is *always* prudent to check the value of the return code if the called program isn't known or trusted:

```
         - - -                     Set up registers for linkage
         BASR  14,15               Link to subroutine as usual
         CHI   15,0                Check for return code 0
         JE    Ret000              Return code = 0
         CHI   15,4                Check for return code 4
         JE    Ret004              Return code = 4
         CHI   15,8                Check for return code 8
         JE    Ret008              Return code = 8
         J     BadRetCd            Unknown/invalid return code
```

Figure 513. Checking for valid return code values

Subroutines that do very complex operations may need to give more detailed information about the reason for a particular return code. A common convention is to put a *reason code* in GR0, as illustrated in Figure 514 on page 781:

```
BadRet   L     13,Save+4          Retrieve addr of caller's save area
         L     0,ReasonCd         Load the reason code into GR0
         L     14,12(,13)         Restore the return address
         LM    1,12,24(31)        Restore GR1 through GR12
         LA    15,4               Set return code to +4
         OI    15(13),X'01'       Set return flag in caller's area
         BR    14                 Return to caller
ReasonCd DC    X'0022006D'        Reason code for a specific problem
```

Figure 514. Setting a reason code in register 0

As with the SAVE macro instruction, a RETURN macro is available in the macro library for use by Assembler Language programs. You usually use one of these forms (where the name-field entry name is optional):

```
name     RETURN  (R1,R2)             Restore registers
name     RETURN  (R1,R2),T           Restore regs, set return flag
name     RETURN  (R1,R2),T,RC=nn     Restore regs, set flag and retcode
name     RETURN  (R1,R2),T,RC=(15)   Restore regs, set flag and retcode
```

The first operand generates an LM instruction to reload registers R_1 through R_2, assuming that GR13 contains the address of the caller's save area before the RETURN macro instruction is executed. The second operand ("T") causes the return flag to be set, and the third operand causes the value "nn" or the existing contents of GR15 to be used as the return code. The second operand can be omitted in the third and fourth forms of the RETURN macro instruction.

We can rewrite Figure 509 on page 779 to use RETURN macros:

```
*        - - -                    Valid result in GR0
GoodRet  L     13,Save+4          Restore caller's save area address
         RETURN (14,12),T,RC=0    Restore registers, etc, RC = 0
BadRet   L     13,Save+4          Restore save area pointer
         RETURN (14,12),T,RC=4    Restore registers, set RC = 4
```

Figure 515. Using RETURN macros to set return flags and return codes

The RETURN macros put the specified return codes in GR15, and then reload GR14 and the other registers.

Sometimes, an alternate return branch does not put a return code in GR15, but returns instead at the equivalent offset from the address in GR14. A revision of Figure 509 on page 779 shows how this might be done:

```
*        - - -                    Valid calculated result is in GR0
GoodRet  L     13,Save+4          Retrieve pointer to caller's area
         LM    14,12,12(13)       Restore GR14 through GR12
         OI    15(13),X'01'       Set return flag
         BR    14                 Return to caller
*        - - -                    Invalid argument, no result in GR0
BadRet   L     13,Save+4          Retrieve addr of caller's save area
         LM    14,12,12(13)       Restore all the registers
         OI    15(13),X'01'       Set return flag
         B     4(,14)        ←─   Return to caller at offset +4
```

Figure 516. Returning to an error branch without a return code

For this to work, the caller must have placed enough 4-byte branch instructions after the BAS or BASR, as in Figure 517 on page 782. Unlike Figure 511 on page 780, caller and callee must agree which method will be used to handle error returns.

```
         - - -
         BAS   14,MySub           Call my subroutine
         B     Ret000             Return code 0, no problems
         B     Ret004             Return code 4, minor problem
         B     Ret008             Return code 8, middling problem
         B     Ret012             Return code 12, major problem
         - - -
```

Figure 517. Call with error branch instructions

If the called routine returns at a larger offset (say, 16) in Figure 517 than the calling routine expected, control would be given to something unexpected.

37.5.5. Conventions for Floating-Point Registers

If a called subroutine modifies the floating-point registers, it is responsible for saving some of them: FPRs 0-7 are considered "volatile" and their contents need not be preserved, but the contents of FPRs 8-15 must be saved and restored across calls. That is, the *called* routine must save and restore modified non-volatile FPRs; this allows you to optimize certain variables into non-volatile registers across calls.

37.5.6. Main-Program Parameters

When you invoke a main program you often pass parameters, typically in the PARM field of a JCL statement:

```
//       EXEC  PGM=MYPROG,PARM='YES,NO,UP,DOWN'
```

The string of characters is passed to your program as shown in Figure 518:

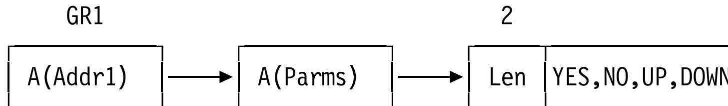


Figure 518. Convention for passing main-program parameters

where the halfword **Len** contains the number of bytes in the following character string.

One of three methods may be used to indicate there are no parameters for the main program:

- The address in GR1, "A(Addr1)" may be zero.
- The address pointed to by GR1, "A(Parms)" may be zero.
- The halfword length **Len** may be zero.

Exercises

37.5.1.(4) In the early days of System/360, all programs executed in 24-bit addressing mode, and a return flag was set with this instruction:

```
         MVI   12(13),X'FF'       Set return flag
```

which set the high-order byte of the return address to all 1-bits. Why could the high-order byte of the return address not be all 1-bits resulting from a normal subroutine call? (You may have to do some historical research to answer this question!)

37.5.2.(2)+ A programmer decided to use this calling sequence to his subroutine:

```
         CNOP  2,4                Align to middle of fullword
         LA    13,MySave          Point GR13 to my save area
         L     15,=A(SUB)         Set GR15 to entry point address
         LA    14,Return          Set GR14 to return address
         BASR  1,15               GR1 points to parm list; call SUB
         DC    A(P1,P2,P3)        Argument address list, with
         DC    AL3(P4+X'80000000')     ... variable-length flag bit
Return   NOP   999                Calling point identifier
```

Does this sequence conform to standard calling conventions?

37.5.3.(3)+ The instructions in Figures 511 and 512 on page 780 assume that the return code in GR15 is a nonnegative multiple of 4. Assuming that the return code must be less than or equal to 12, revise the instructions as follows:

- If the return code is negative, set it to 12.
- If the return code is greater than 12, set it to 12.
- If the return code is not a multiple of 4, round it to the next higher multiple of 4.

37.5.4.(3) Write a subroutine named **WHO** that determines if there is an entry point identifier at the entry point of *its* caller, and print it if there is one. For example, if **WHO** was called from **WHAT**, it might print

```
WHO called from WHAT
```

37.5.5.(3) Write a subroutine **WHO2** (as in Exercise 37.5.4) that will *also* print the caller's calling point identifier if it is present.

37.5.6.(2)+ The technique for using return codes illustrated in Figure 512 on page 780 is unsafe if the called routine does not place a valid return code in GR15. Rewrite the code sequence so that valid return codes 0, 4, 8, and 13 will branch as desired, and invalid codes will cause a branch to **InvCode**.

37.5.7.(1) What is the maximum possible value of a calling point identifier?

37.5.8.(1)+ A programmer suggested this instruction sequence for returning to a caller with a return code in GR15. Will it work? Explain.

```
         L     13,4(,13)          Restore caller's save area addr
         MVC   16(4,13),RetCode   Move my return code to GR15 slot
         LM    14,12,12(13)       Restore registers
         BR    14                 Return to caller
```

37.5.9.(2)+ Show how you can use a based branch to branch into a branch table (as in Figure 512 on page 780) without requiring a base register.

37.5.10.(1) Show a way to test whether a return code is a multiple of four. If it isn't, branch to **BadRC**.

37.5.11.(2)+ A programmer claimed that you can test whether the return code in GR15 is a multiple of 4 using these instructions:

```
Choice   CLI   *+1,B'11111100'
         EX    15,*-4
         BNE   Error              Branch if not a multiple of 4
```

Prove or disprove his claim, and explain your proof.

37.6. Assisted Linkage (*)

Up to now we have discussed *direct linkage:* the calling program transfers control directly to the subroutine's entry point. It is sometimes more convenient to use an *indirect* or *assisted linkage:* instead of branching directly to the called program, control passes to an intermediate assisting routine to do some useful activities (such as tracing, status preservation, and counting) and then branch to the subroutine.

This technique can relieve caller and callee of much of the labor associated with subroutine linkages, but it requires careful planning of conventions used to communicate with the intermediate assisting routine.

To illustrate, suppose **ShftRt** and **Print** are two subroutines called in a complex program. To call a routine, we place its “subroutine number” into GR15 and then link to an internal **Caller** routine that eventually branches to the desired subroutine. We can create the “subroutine numbers” by appending a “#” character to the actual routine name, and defining Equ statements to generate the number. Figure 519 shows how you might use an assisted linkage:

```
         - - -
         LA    15,ShftRt#        Load subroutine number for ShftRt
         BAS   14,Caller         Link to ShftRt via Caller routine
         - - -
         LA    15,Print#         Load subroutine number for Print
         BAS   14,Caller         Link to Print routine via Caller
```

Figure 519. Example of calling with assisted linkage

A simple form of **Caller** could be written as in Figure 520:

```
Caller   L     15,AdrTbl(15)     Get true subroutine address
         BR    15                Branch to subroutine
AdrTbl   DC    0A(0)             Align start of table
ShftRt#  Equ   *-AdrTbl          Subroutine number for ShftRt
         DC    A(ShftRt)         True address of ShftRt
Print#   Equ   *-AdrTbl          Subroutine number for Print
         DC    A(Print)          True address of Print
         - - -
```

Figure 520. Example of a routine to implement assisted linkage

The called routine will not be aware of the indirect nature of the call, and the return from callee to caller will be direct, bypassing the linkage routine.

This scheme apparently wastes instructions in getting from one routine to another. However, by expanding information kept by the **Caller** routine it can, for example, display the name and address of the routine being called and of its caller, the addresses of the arguments, the return address, and so forth.

For example, the **Caller** routine could keep a count of the number of times each routine is called. Figure 521 shows a way to do this; we assume that no base registers are available to provide addressability in the **Caller** routine.

```
Caller   ST    14,12(,13)        Save GR14 in caller's area
         LARL  14,AdrTbl         Set c(GR14) = A(AdrTbl)
         AR    14,15             Form address of table entry
         L     15,4(,14)         Get count for called routine
         AHI   15,1              Add 1
         ST    15,4(,14)         Restore count
         L     15,0(,14)         Load true entry point address
         L     14,12(,13)        Restore caller's return address
         BR    15                Branch to chosen subroutine
AdrTbl   DC    0A(0)             Align start of table
ShftRt#  Equ   *-AdrTbl          Subroutine number for ShftRt
         DC    A(ShftRt)         Address of ShftRt
         DC    F'0'              Count of calls to ShftRt
Print#   Equ   *-AdrTbl          Subroutine number for print
         DC    A(Print)          Address of print
         DC    F'0'              Count of calls to Print
         - - -
```

Figure 521. Assisted linkage routine with counters

In large programs this *tracing* information can be extremely helpful in finding errors in logic and flow of control. The routines called using assisted linkages may have been tested, but difficulties always appear when smaller routines are combined into large programs.

The value of such a tracing and diagnostic tool often far outweighs its minor costs; and the overheads of the indirect subroutine linkage can be minimized by setting a flag to indicate whether or not traces and diagnostics are wanted, or if the indirect call should go directly to the chosen routine.

Exercises

37.6.1.(2)+ Assuming each of the called routines in Figure 521 on page 784 has an entry point ID, show the instructions you'd add to the **Caller** routine to load GR1 and GR2 with the address and length of the *called* routine's ID.

37.6.2.(4)+ Assuming each of the called routines in Figure 521 on page 784 has an entry point ID, show the instructions you'd add to the **Caller** routine to load GR1 and GR2 with the address and length of the ID of the *calling* routine.

37.6.3.(2)+ In Figure 521 on page 784 the first instruction saves a register in the caller's save area. Will it matter whether or not the called routine (that eventually receives control from **Caller**) also executes the same instruction?

37.6.4.(2)+ Suppose the call to the **Caller** routine is changed to pass the subroutine number as a halfword following the BAS instruction:

```
         BAS   14,Caller          Link to 'Caller' routine
         DC    Y(ShftRt#)         With inline subroutine number
         - - -                    Return here from called routine
```

The **Caller** routine could be modified like this:

```
         LA    15,AdrTbl          Address of zero-th target routine
         AH    15,0(0,14)         Add subroutine number to GR15
         L     15,0(0,15)         Load correct entry point address
         AHI   14,2               Increment return address by 2
         BR    15                 Enter called routine
AdrTbl   DC    A(ShftRt)          True address of ShftRt
         DC    A(Print)           True address of Print
         - - -                    ...etc...
```

Show how the interface could be slightly modified to not require incrementing the address in GR14 by two.

37.6.5. Revise the instruction sequences in Exercise 37.6.4 to pass the subroutine number in GR0 to the **Caller** program.

37.7. Lowest Level Subroutines

If a routine makes no calls to lower-level routines (*including* calls to I/O or other Supervisor services), it need not contain a save area.

Suppose we need a subroutine that sets a string of bytes to zero; we will write a lowest level subroutine **ClearMem** that will zero up to 256 bytes at a time. Its first argument gives the address of the first byte of the area to be cleared, and the second is the address of a positive fullword integer specifying the number of bytes.

```
         Using *,15                  Caller will preset GR15
ClearMem STM   14,2,12(13)           Save GR14 through GR2
         XC    8(4,13),8(13)         Zero the caller's forward chain
         LM    1,2,0(1)              Get argument addresses
         L     2,0(0,2)              Load number of bytes to zero
         BCTR  2,0                   Decrement by 1 for EX instruction
         EX    2,Zero                Execute XC instruction
         LM    1,2,24(13)            Restore modified registers
         BR    14                    Return to caller
Zero     XC    0(0,1),0(1)           Clear a variable-length area
         Drop  15                    Base in GR15 no longer available
```

Figure 522. Example of a lowest level subroutine

Because no calls are made by this routine, GR15 can be used as its base register. The only registers changed are GR1 and GR2, which are restored on return.

We could have saved only GR1 and GR2, but it's good practice to *always* save registers GR14 and GR15 in the caller's save area. Because those registers define the addresses from and to which the call is made, those addresses in the save area can help diagnose difficulties if a program goes astray.

The technique shown in Figure 522 should be used only if you are absolutely certain that the routine will never be modified to call another lower-level routine. If that routine causes an interruption, debugging your program will be much more difficult.

Exercises

37.7.1.(2) Assume that the **ShftRt5** routine in in Figure 489 on page 766 is a lowest level routine that returns its result in GR0. Revise it to save and restore the minimum necessary number of general registers.

37.7.2.(3)+ Write a **RanInt** subroutine that calculates random integers according to the formula

$$Xnew = A \times Xold \pmod{p}$$

where A = 16807 and p = $2^{31}-1$ = 2147483647. That is, given a previous random integer Xold, Xnew is the remainder of (A×Xold)/p. The subroutine should conform to these conventions:

- The return address is in GR14.
- GR13 contains the address of a save area.
- The entry point instruction should be named **RanInt**.
- When control reaches the entry point its address will be in GR15.
- The value of Xold is passed to the routine in GR0.
- The new random number Xnew is returned to the caller in GR0.
- Restore all registers except GR0 to their original contents.

Note: The quality of the random sequences generated by this routine is not high enough for serious or lengthy simulations.

37.7.3.(2) Suppose in Figure 522 that the LM instruction to restore the registers had been written with operand field entry `1,12,24(13)` instead of `1,2,24(13)`. Describe what differences this (not unusual) programming error might make in the behavior of the program.

37.8. Summary

This section has discussed topics that can help you divide a complex application into smaller parts that can be assembled and tested individually. We've seen conventions used for

- the registers used for standard linkage conventions
- techniques used for argument passing, including variable-length argument lists
- standard save areas
- status preservation
- useful instructions supporting these activities.

Internal subroutines[246] can use any convenient convention for linkage and argument passing that are agreed between caller and callee, and used consistently. These “local” conventions can be simple and efficient, but subroutines using them can be difficult to extract for use in other programs.

Subroutines using standard linkage conventions can be written once, assembled, and installed in a “Library of object and load modules” (see Figure 24 on page 73) for use by many programs. Reusing existing subroutines makes it easier to share code, and simplifies and speeds application development.

37.8.1. Standard Linkage Conventions

This is a summary of the standard linkage conventions used in many operating system environments on System z.

1. The entry point address is in general register 15 on entry to the called routine.
2. The return address is in general register 14 on entry to the called routine.
3. The address of the list of argument addresses is in general register 1 on entry to the called routine. If there are no arguments, general register 1 is set to zero.

   For calls to routines using 32-bit argument addresses and in which the number of arguments is known to be fixed and invariable, it is not necessary to set the high-order bit of the last argument address (but it's always a good practice!).
4. For calling programs using only 32-bit general registers, the address of an 18-word save area is in GR13 on entry to the called routine.
5. The setting of the Condition Code on entry to the called routine need not be preserved.
6. Subroutines that calculate an integer value often return it in general register 0.
7. The contents of floating-point registers 0-7 need not be preserved by the called routine; if any of floating-point registers 8-15 will be used their contents must be saved and restored before returning to the caller. That is, FPR0-FPR7 are considered “volatile”.

   Routines that calculate floating-point values often return them in FPR0 or (FPR0,FPR2).
8. General registers 2 through 14 must be saved by the callee and restored to their original values before control is returned to the caller. (In general, it is best to restore the contents of *all* registers not containing return values whenever practical.) Some operating system services may not restore all registers; be sure to check the system's documentation.

   If the called routine uses 64-bit registers, it must save and restore the low halves of general registers 2-13 and the high halves of general registers 2-14.

   Many Operating System services expect GR13 to point to a standard save area, and may not restore general registers 14, 15, 0, and 1.
9. In the Floating-Point Control Register, preserve the mask and rounding bits. The flag bits and Data Exception Code (DXC) need not be saved.

---

[246] More precisely: *internal subroutines within a single control section.* We'll see why this is important in Section 38, when we discuss separately assembled routines.

10. Return codes, if used, are integer multiples of 4 in GR15 on return.

    Sometimes more detailed information is returned than a single integer in GR15 can provide; if so, a reason code is often returned in general register 0.

11. For programs executing in 64-bit addressing mode, the caller must provide a 144-byte doubleword aligned Format-4 save area with the characters 'F4SA' in the second word.

Other conventions are used for identifying entry and calling points, and a return flag.

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode	Mnemonic	Opcode
BAS	4D	BRAS	A75
BASR	0D	BRASL	C05

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic	Opcode	Mnemonic
0D	BASR	A75	BRAS
4D	BAS	C05	BRASL

Terms and Definitions

argument
A value supplied by a calling program.

calling point identifier
A NOP instruction with a halfword identifying number in place of its addressing halfword, of the form X'4700nnnn'.

entry point
The first instruction to receive control when a routine is invoked.

entry point identifier
A string of characters following the first instruction at an entry point, providing descriptive information about the entry.

linkage convention
An agreed set of rules for transferring control between a calling and a called program, passing arguments and receiving results, and preserving caller information during the execution of the called program so it can be restored on return to the caller.

parameter
A place-holder in a called program, to be assigned a value from an argument provided by a calling program.

return address
The address of an instruction to which control should be passed when a called routine completes its execution.

return code
A small integer value (usually a multiple of 4), often placed in GR15 prior to returning to a calling program.

Programming Problems

Problem 37.1. Write a program to print the 168 prime numbers less than 1000. Create an internal subroutine to convert a number from its internal binary representation to character form and print it.

Problem 37.2. Write a subroutine named PD2CH with two arguments: the first (input argument) is a valid packed decimal number 6 bytes long, and the second (output argument) is a character string 12 bytes long containing a blank character followed by the digits of the result. Your routine should replace leading zero digits by blanks.

Problem 37.3. Do as in Problem 37.2, but now write a subroutine named PD2CHN that accepts two input arguments: the packed decimal number and its length in bytes. Use the second argument to determine the minimum necessary length of the third (output) argument.

Problem 37.4.(3)+ Write a subroutine named **GCD** that calculates the greatest common divisor of two positive 32-bit integer arguments, using standard linkage conventions. The subroutine should accept either two or three arguments:

- The first two arguments are the integers whose GCD is to be found. In this case, return their GCD in GR0.
- If a third argument address is provided, store the GCD at that address.

The greatest common divisor function GCD(x,y) can be defined by

GCD(x,y) = **if** x=0 **then** y **else** GCD(remainder[y/x],x).

Restore all registers other than GR0 and GR1.

Problem 37.5.(2)+ Write a subroutine with entry name **PMax** with a single argument: a pointer in GR1 to a list of addresses. Each address in the list points to a packed decimal number preceded by a single byte containing the length in bytes of the packed decimal number, as in

```
         DC    AL1(3),P'1234'
```

Only the address of the last item in the list has its high-order bit set to 1.

The subroutine should return in GR1 the address of the list item with the largest magnitude. None of the data in the list may be modified.

38. Large Programs, Control Sections, and Linking

```
 3333333333     8888888888
333333333333   888888888888
33        33   88        88
          33   88        88
          33    88      88
       3333      88888888
       3333      88888888
          33    88      88
          33   88        88
33        33   88        88
333333333333   888888888888
 3333333333     8888888888
```

As programs grow, we need to consider (at least) three issues:

1. How to provide addressability for parts of the program that require it.
2. How to subdivide a program into smaller, more manageable pieces
   - within a single assembly,
   - with separate assemblies.
3. How to combine the separate pieces into a complete application.

Thus far, we've discussed programs where a single base register was enough to provide addressability for the entire program. But programs frequently grow large enough that more than one base register might be needed.

We'll consider two techniques for providing addressability in large programs:

1. We can allocate enough base registers to ensure that every byte of the program can be addressed; we call this "uniform" addressability.
2. We can rearrange parts of the program into segments that require addressability, and segments that don't.

38.1. Uniform Addressability for Large Programs

If two or three general registers can be assigned as base registers for the duration of the program, we use some simple techniques.

For the following examples, suppose we need three base registers to provide addressability for the entire program, and that we have assigned GR10, GR11, and GR12 for this purpose. (Three base registers can provide uniform addressability for up to 12K bytes.) A typical instruction sequence for initializing these base registers is shown in Figure 523 on page 791.

```
PROG     Start 0                   Start LC at zero
         BASR  10,0                Establish first base register
         Using BaseLoc,10          Inform assembler
BaseLoc  LAY   11,4096(,10)        Create c(GR10)+4096 in GR11
         Using BaseLoc+4096,11     Inform assembler of 2nd base
         LAY   12,8192(,10)        Create c(GR10)+8192 in GR12
         Using BaseLoc+8192,12     And inform the assembler again
```

Figure 523. Establish three base registers (1)

The USING statements need not appear exactly where they are placed in Figure 524. Because none of the LAY instructions contains an implied address, we can place all the USING statements after the BASR:

```
PROG     Start 0
         BASR  10,0                Set first base register
         Using *,10                Addressability based on GR10
         Using *+4096,11           Addressability based on GR11
         Using *+8192,12           Addressability based on GR12
         LAY   11,4096(0,10)       Create second base
         LAY   12,8192(0,10)       And third base
```

Figure 524. Establish three base registers (2)

Or, we can place them after the LAY instructions:

```
PROG     Start 0
         BASR  10,0
BaseLoc  LAY   11,4096(0,10)       Create second base
         LAY   12,8192(0,10)       And third base
         Using BaseLoc,10          Addressability based on GR10
         Using BaseLoc+4096,11     Addressability based on GR11
         Using BaseLoc+8192,12     Addressability based on GR12
```

Figure 525. Establish three base registers (3)

In situations like this it's easier to use a more general form of the USING statement. Instead of writing a single register number after the first operand, we can write up to 15 register numbers:[247]

Using value, $regnum_1$, $regnum_2$, ..., $regnum_k$

When the Assembler processes this statement, the entries made in the Using Table (described in Section 10.8 on page 125) are identical to the entries that would be made from these statements:

Using value, $regnum_1$
Using value+4096, $regnum_2$
Using value+8192, $regnum_3$
- - -
Using value+(k-1)*4096, $regnum_k$

That is, each register specified in the third and following operands is assumed to contain a value 4096 larger than the value assigned to the previous register. We could therefore have written Figure 524 in the simpler form shown below.

[247] A program needing 15 base registers for addressability wouldn't be able to do much with the one remaining register! We'll see that relative addressing can help a lot.

```
PROG     Start 0
         BASR  10,0              Establish base in r10
         Using *,10,11,12        Using Table entries for GR10/GR11/GR12
         LAY   11,4096(0,10)     c(GR11) = c(GR10)+4096
         LAY   12,8192(0,10)     c(GR12) = c(GR10)+8192
```

Figure 526. Establish three base registers (4)

Similarly, Figure 525 on page 791 could have been written this way:

```
PROG     Start 0
         BASR  10,0
BaseLoc  LAY   11,4096(0,10)
         LAY   12,8192(0,10)
         Using BaseLoc,10,11,12
```

Figure 527. Establish three base registers (5)

You can also use arithmetic instructions with immediate operands instead of the LAY instructions:

```
PROG     Start 0
         BASR  10,0
BaseLoc  LR    11,10
         AHI   11,4096
         LR    12,11
         AHI   12,4096
         Using BaseLoc,10,11,12
```

Even though the same number of bytes are generated, two more instructions must be executed.

38.1.1. Other Techniques (*)

You can create the base addresses in GR11 and GR12 shown in Figures 523 to 527 in other ways. If a binary halfword integer constant 4096 is in an area named **HW4096**, we might write the following:

```
PROG     Start 0
         BASR  10,0              Establish first base register
         Using *,10,11,12        Make entries in using table
         LR    11,10             Copy base to GR11
         AH    11,HW4096         Add 4096 to old value
         LR    12,11             Copy second base to GR12
         AH    12,HW4096         Add 4096 to get third base
```

Figure 528. Establish three base registers with risks (6)

This example contains a serious pitfall: since we do not necessarily know where the constant named **HW4096** is located, we also do not know which base register will be chosen by the Assembler when the implied addresses of the AH instructions are resolved into base-displacement form! Indeed, the Assembler has been instructed by the USING statement to assume that R11 and R12 may be used for addressing at a point in the program which logically *precedes* the point where the necessary addresses have been placed in the registers. Thus, we might find that the Assembler has resolved the implied addresses in a way that leads to execution-time errors. (See Exercises 38.1.5 and 38.1.14.)

To be safe, we must place the constant in a part of the program where we know it will be addressed relative to a register whose contents at execution time have been correctly established.

```
PROG     Start 0
         BASR  10,0              GR10 will be okay as our base reg
         Using *,10,11,12
         LR    11,10             Move base to GR11
         AH    11,HW4096         Add 4096 to get second base
         LR    12,11             Move second base to GR12
         AH    12,HW4096         Add 4096 more for 3rd base
         J     Next              Jump over the constant
HW4096   DC    H'4096'           (addressed by GR10 as base)
Next     - - -                   ...rest of program...
```

Figure 529. Establish three base registers (7)

We can also load base addresses into registers using *address constants*.[248] Suppose we define two fullword address constants with the statement

```
Addrs    DC    A(BaseLoc+4096,BaseLoc+8192)
```

We will see in Section 38.7.2 how the *true* execution-time addresses corresponding to the expressions “BaseLoc+4096” and “BaseLoc+8192” are placed into the two fullword locations when the Supervisor loads our program into memory for execution. Then we could write

```
PROG     Start 0
         BASR  10,0              Set initial base register
         Using *,10,11,12
BaseLoc  LM    11,12,Addrs       Load GR11 and GR12
```

Figure 530. Establish three base registers (8)

Problems can also arise here if the constants at **Addrs** are far enough away from **BaseLoc** that the value of the implied address of **Addrs** causes the Assembler to resolve the addressing halfword of the LM instruction with GR11 or GR12 as the base register *before* its value has been established! To be safe, we should rewrite Figure 530 to be sure that the symbol **Addrs** will be addressable only by GR10.

```
PROG     Start 0                 Set value of location counter
         BASR  10,0              Establish first base
         Using *,10,11,12        Make Using Table entries
BaseLoc  LM    11,12,Addrs       Set GR11 and GR12 from adcons
         J     Next              Jump over constants
Addrs    DC    A(BaseLoc+4096,BASELoc+8192)
Next     - - -                   ...continuation of program...
```

Figure 531. Establish three base registers (9)

In Section 38.2 we will examine cases where it is either undesirable or impossible to use the above techniques for maintaining addressability throughout an entire program.

Exercises

38.1.1.(2)+ In Figure 525 on page 791, why could we not write

```
Using *,10
Using *+4096,11
Using *+8192,12
```

as in Figure 524 on page 791?

38.1.2.(2)+ In Figure 525 on page 791 and Figure 527 on page 792, why isn't at least one USING needed immediately following the BASR?

[248] Or, *adcons*, for short. We first encountered them in Section 12.2, where we noted that the expression in an A-type constant may be relocatable or absolute.

38.1.3.(3)+ The addressing range of an instruction with a signed 20-bit displacement is about ±512K bytes. Instead of the three base registers in Figures 523 to 525 ending on page 791, why can't we write something like this, to be able to address over a million bytes with just one base register?

```
Prog     Start 0
         BASR  10,0
BaseLoc  LAY   10,512000(,10)     Establish base register
         USING BaseLoc+512000,10  Provide lots of addressability
         - - -
```

The next three exercises illustrate suggested methods for providing uniform addressability using address constants. Each contains an error for you to find.

38.1.4.(2)+ What's wrong?

```
Prog     Start 0
Begin    BASR  3,0
         Using *,3,4,5
         LM    4,5,=A(Begin+4096,Begin+8192)
```

38.1.5.(2)+ What's wrong?

```
Prog     Start 0
Begin    BASR  3,0
         Using *,3,4,5
         LM    4,5,=A(Begin+4096,Begin+8192)
         B     Exec
         LTORG
Exec     EQU   *
```

38.1.6.(2)+ What's wrong?

```
Prog     Start 0
Begin    BASR  3,0
         Using *,3,4,5
         LM    4,5,BaseAdr
         B     Exec
BaseAdr  DC    A(Begin+4096,Begin+8192)
Exec     EQU   *
```

38.1.7.(2) A programmer was asked the following question: “Suppose your program is ‘covered’ by base registers 2, 3, and 4. A part of the code requires register 2 for another purpose. Re-establish register 2 as a base after that computation is finished.” Criticize his proposed solution below, assuming GR2 originally contained the address of **Origin**.

```
         - - -
         DROP  2                  Don't use GR2 as a base
         - - -
         - - -                    Do arithmetic with GR2
         - - -
         LA    2,Origin           Restore GR2 to original value
         Using Origin,2           And re-issue USING
```

38.1.8.(2)+ In Figure 531 on page 793, what would happen if the statement

```
         Using    *,10,11,12
```

is placed *after* the LM instruction?

38.1.9.(2)+ You have written a subroutine with entry point **SubRtn** and you know that GR15 will contain its address when control arrives there. Give an example to illustrate and explain why the following USING statement may be incorrect:

```
         Using *,15
SubRtn   ST    2,SaveR2          First instruction of SubRtn
         - - -                   Rest of subroutine
```

38.1.10.(2)+ A programmer initialized his base registers with these instructions. Why might this not work?

```
Prog     Start 0
         Using *,15
         LM    11,12,=A(Prog,Prog+4096)
         Using *,11,12
         Drop  15
```

The instruction sequences in Exercises 38.1.11 through 38.1.19 were proposed as solutions to the problem of initializing multiple base registers at the beginning of a program; find their errors. (If the errors are hard to find, try assembling each, study the assembled code, and calculate what would actually appear in the designated registers when the instructions were executed.)

38.1.11.(1)+ What's wrong?

```
Prog     Start 0
         BAS   10,0
         Using *,10,11
         LA    11,Prog+4096      What object code is generated?
```

38.1.12.(1)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using *,10,11
         LR    11,10
         AH    11,=H'4096'
```

38.1.13.(2)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using *,11
         LA    11,Prog+4096
```

38.1.14.(1)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using *,10,11,12
         LA    11,Prog+4096
         LA    12,Prog+8192
```

38.1.15.(1)+ What's wrong?

```
Prog     Start 0
         BASR  2,0
Here     Using *,2,3,4,5
         LA    3,Here+4096
         LA    4,Here+8192
         LA    5,Here+12288
```

38.1.16.(1)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using *,10,11
         LA    11,*+4096
```

38.1.17.(1)+ What's wrong?

```
Prog     Start 0
Here     BASR  10,0
         Using Here,10,11
         LA    11,Here+4096
```

38.1.18.(1)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using Here,10,11
Here     LA    11,Here+4096
```

38.1.19.(2)+ What's wrong?

```
Prog     Start 0
         BASR  10,0
         Using *,10
         LA    11,4095(,10)
         LA    11,1(,11)
         Using *+4096,11
```

38.1.20.(2) A programmer decided not to use LAY instructions in Figure 524 on page 791, and wrote

```
         BASR  10,0
         Using *,10
         Using *+4095,11
         Using *+8190,12
         LA    11,4095(,10)
         LA    12,4095(,11)
```

Is there anything wrong with this method? If so, what and why?

38.1.21.(2)+ Suppose the program segment in Figure 528 on page 792 is rewritten as follows:

```
Prog     Start 0
         BASR  10,0
         Using *,10
BaseLoc  LR    11,10
         AH    11,HW4096
         Using BaseLoc+4096,11
         LR    12,11
         AH    12,HW4096
         Using BaseLoc+8192,12
```

If the halfword area of memory named **HW4096** actually contains `X'1000'`, and the program is less than 10,000 bytes long, will this segment work correctly? Explain your answer.

38.1.22.(2)+ In Figure 529 on page 793, what would happen if the operand of the DC instruction is changed to H'4000'? If it is changed to H'5000'? What changes might be needed to the other instructions?

38.1.23.(3) In Figure 528 on page 792, a risky method for initializing base registers is illustrated. Assuming that the value of the symbol **HW4096** is `X'1234'`, determine (1) the bases and

displacements assigned to the implied addresses of the AH instructions, and (2) the addresses that would be found in GR10, GR11, and GR12 if the program is loaded into memory beginning at address X'20000', and the instructions in the program segment are executed. (That is, the BASR is at address X'20000'.) Then repeat the above two calculations, on the assumptions that the value of the symbol **HW4096** is X'2234', and then X'3234'. Explain your results.

38.1.24.(2)+ In Figure 528 on page 792, show how to use immediate operands to eliminate the memory references to the constant named **HW4096**.

38.1.25.(2)+ Revise Figure 523 on page 791 through Figure 529 on page 793 to use arithmetic-immediate instructions.

38.1.26.(2) Modify the instructions in Figure 526 on page 792 to use LA instructions instead of LAY instructions.

38.1.27.(1) Modify the instructions in Figure 531 on page 793 to initialize *four* base registers rather than three.

38.1.28.(2)+ A programmer suggested using the following instructions instead of those in Figure 530 on page 793:

```
PROG     Start 0                 Set location counter
         BASR  10,0              Establish temporary base
         Using *,10              Provide addressability
         LM    10,12,Addrs       Set GR10, GR11, and GR12 values
         BR    10                Branch over the constants
BaseLoc  DC    A(BaseLoc,BaseLoc+4096,BaseLoc+8192)   Base addresses
         Drop  10
         Using BaseLoc,10,11,12  Set program addressability
BaseLoc  - - -
```

Will they work correctly? Might they have any advantages or disadvantages compared to Figure 530 on page 793?

38.2. Simplifying Addressability Problems in Large Programs

As programs (inevitably) grow larger, more base registers may be needed to address the entire program. For programs of reasonable size, this is usually not a problem. However, if a program becomes large enough that many registers are required for addressability and few are left for the "business" of the program, we need to find ways to simplify the program.

First, we'll look at cases where enough base registers are available; then we'll discuss several techniques that can be used to reduce the number of required base registers.

38.2.1. Internal Subroutines Without Local Addressability

Most often, internal subroutines can use the "global" addressability provided for the entire program.

Here is a **ShftRt** subroutine like the one we saw in Figure 477 on page 760:

```
*        ShftRt Arguments in registers 0 and 1
ShftRt   LTR   1,1               Test value of N
         BNM   ShftOK            Branch if nonnegative
         SR    1,1               Set N to zero if it was minus
ShftOK   SRL   0,2(1)            Shift (2+max(N,0)) bit positions
         BR    14                Return to caller
```

Because it contains an implied address reference, it requires local addressability. But if the containing program has provided uniform addressability for the entire program, we need no local base register(s) to provide addressability for the reference to the symbol **ShftOK**.

Occasionally we can rewrite a subroutine by replacing a based branch instruction with a relative branch, and no base register may be needed.

```
*        ShftRt Arguments in registers 0 and 1
ShftRt   LTR   1,1                Test value of N
         JNM   ShftOK             Branch relative if nonnegative
         SR    1,1                Set N to zero if it was minus
ShftOK   SRL   0,2(1)             Shift (2+max(N,0)) bit positions
         BR    14                 Return to caller
```

We'll see more examples of this technique.

Internal subroutines typically use either the standard linkage conventions described in Section 37.2 on page 765, or local conventions specific to the needs of the internal subroutine. Figure 532 shows an unusual way to call a subroutine that doesn't need a local base register.

```
         L     0,Logic            Place arguments in GR0 ...
         L     1,Shift            ...and in GR1
         L     14,AdShft          Subroutine address in GR14 now
         BASR  14,14              Funny branch to subroutine
         ST    0,Result           Store answer
         - - -                    ...and move on
Logic    DS    F                  Logical datum
Shift    DS    F                  Shift amount
Result   DS    F
AdShft   DC    A(ShftRt)          Address of subroutine
```

Figure 532. Calling a subroutine not needing local addressability

The unusual aspect of this program segment is that GR14 (not GR15!) is used to contain (briefly) the entry point address, and also the return address!

While this technique is not very general, it can be useful when the number of available registers is severely limited.

38.2.2. Internal Subroutines With Local Addressability

Some programs are so big it is impractical to provide addressability for the entire program. We will look at simple ways to divide a large program into smaller and more manageable parts, starting with internal subroutines.

Suppose the **ShftRt** subroutine in Figures 477 through 481 starting on page 760 is part of a large program, and we know neither

- where the subroutine will be placed relative to the rest of the program, nor
- whether addressability will be available in the subroutine.

For now, we'll assume that the arguments are passed from the caller to the subroutine in registers GR0 and GR1, as in Figures 477 and 478.

Because the caller can't always be sure the first instruction of the subroutine is addressable, he can't safely call it using a

```
         BAS   14,ShftRt
```

instruction as before. If the address of the instruction named **ShftRt** is available, we can load the address into a register and branch to the subroutine. If the caller and the subroutine agree that GR15 will contain the address of the first executed instruction of the subroutine (its *entry point* address), we could write

```
         L     0,Logic          Get datum to be shifted in GR0
         L     1,Shift          Shift amount goes in GR1
         L     15,AdShft        Load GR15 with subroutine address
         BASR  14,15            Branch to subroutine
         ST    0,Result         Store shifted result
         - - -                  ...and continue
Logic    DS    F                Logical datum
Shift    DS    F                Shift amount
Result   DS    F
AdShft   DC    A(ShftRt)        Address of subroutine
```

Figure 533. Calling a subroutine not locally addressable

As in Figure 472 on page 757 and those immediately following it, we assumed that GR12 provides addressability for the calling program, so the four fullwords of data are addressable (as we expect from the implied addresses in the Load and Store instructions). Because the instruction named **ShftRt** may not be addressable, we require only that the *address constant* named **AdShft** containing its address be addressable. Also, choosing GR14 and GR15 to contain the return and entry point addresses is a matter of convention; but a convention that assumes that neither GR14 nor GR15 provides addressability in the calling program.

The **ShftRt** subroutine now takes advantage of its address having been put in GR15 by the caller, so we can write it as follows:

```
         Using *,15             Caller will preset GR15
ShftRt   LTR   1,1              Test shift count
         BNM   ShftOK           Branch (based) if not negative
         SR    1,1              Set count to zero
ShftOK   SRL   0,2(1)           Perform shifts
         BR    14               And return to caller
         DROP  15               Nobody should use GR15 hereafter
```

Figure 534. Subroutine with local addressability

We didn't execute a “BASR 15,0” before the USING statement because the USING statement merely tells the Assembler what to assume for assigning bases and displacements. Because the caller places the correct address in GR15 before branching to the subroutine, there is no need to do anything more to establish addressability. Of course, if the caller neglects to preset GR15, the BNM instruction (if taken) might cause a branch into unknown parts of memory; but that's the fault of the caller, since the subroutine is correctly written.

The DROP statement at the end of the subroutine is important: if it's not present, the Assembler must assume GR15 is available for addressing purposes in statements following the subroutine. If GR15 is used as the base register in a subsequent implied address, serious errors could occur, because GR15 may not contain the address of **ShftRt** after return to the caller.

It may seem we haven't gained much flexibility in this simple example. However, remember that the instruction sequence in Figure 534 could have been part of a small test program for checking that the subroutine worked correctly. By extracting the statements of the subroutine and putting the USING and DROP statements before and after them, the subroutine can be put in any convenient part of a larger program.

This is a major reason for using subroutines: you can break your large problem into smaller and more manageable pieces for testing before you combine them into a bigger program.

38.2.3. Minimizing the Number of Base Registers

We'll investigate two basic ideas:

- Replace based-addressing instructions wherever possible with relative-addressing instructions, and replace references to constants and literals by instructions with immediate operands.
- Separate the instructions and data so that the instructions need no base registers to address themselves.

38.2.4. Relative Branches, Immediate Operands, and Long Displacements

First, we can take advantage of the relative branch instructions (and their extended mnemonics) listed in Table 117 on page 330 to replace based branches with relative branches:

```
Before                         After
B     Next                     J     Next
BZ    ZeroVal                  JZ    ZeroVal
BC    12,NoCarry               JC    12,NoCarry
BAS   14,Sub                   JAS   14,Sub
```

Figure 535. Replacing based branch instructions with relative-immediates

Each based branch instruction requires a base register to resolve its Effective Address; relative branches require no base register.

Similarly, each (based) EX instruction can be replaced by its relative-addressing form, EXRL:

```
Before                         After
EX    2,MoveData               EXRL  2,MoveData
```

Figure 536. Replacing a based EXecute instruction with EXRL

Next, we can use the instructions with immediate operands described in Section 21 on page 316 to replace based references to constants and literals:

```
Before                         After
S     8,=F'5000'               AHI   8,-5000
LH    3,=Y(L'Buffer)           LHI   3,L'Buffer
LG    5,=FD'2147483647'        LGFI  5,2147483647
```

Figure 537. Replacing references to constants with immediate operands

Finally, we can use instructions with long 20-bit signed displacements to refer to much more of the program than is possible with instructions using unsigned 12-bit displacements:

```
Before                         After
L     9,=A(SomeData)           LY    9,SomeData
L     9,0(,9)
LA    1,ArgList                LAY   1,ArgList
```

Figure 538. Replacing short unsigned displacements with long signed displacements

Using relative branches and immediate operands, we can replace some instructions with base-displacement addressing that require base registers. However, instructions that load, store, and move data typically require base-displacement addressing.

38.2.5. Separating Instructions and Data

To reduce the number of required base registers, you can put data that will be read, written, or moved into an area addressable with a base register, and put as many of the instructions as possible into an area *not* requiring a base register to address them. For example, this program fragment mixes instructions and data areas, and requires a base register for both instructions and data.

```
         Using *,12              Base register for entire program
         L     2,X
         A     2,Y
         ST    2,Z
         BP    Positive
         - - -
X        DC    F'123456789'
Y        DC    F'7654321'
Z        DS    F                 Some sum
         - - -
Positive L     0,PlusCode        Indicate positive sum
         - - -
PlusCode DC    F'+12'            A small number
```

Figure 539. A program fragment needing reorganization

The program fragment in Figure 539 can be reorganized this way:

```
         Using Data,12       ←— Note the different USING base
         LGFI  2,123456789   ←— Note immediate operand
         AGFI  2,7654321     ←— Note immediate operand
         ST    2,Z
         JP    Positive          Use a relative branch instead
         - - -
Positive LHI   0,+12             Indicate positive sum ←— Moved!
         - - -
Data     DS    0D            ←— Eliminated 3 fullword constants
Z        DS    F                 Some sum
         - - -
```

Figure 540. A program fragment after reorganization

Now, only the data areas following the symbol **Data** are addressable; none of the instructions is addressable — and none needs to be! If this was part of a large program, it's possible we could have eliminated the need for one or more base registers.

It seems logical to put the instructions at the start of the program and the constants and work areas at the end. Because most programs will already have a register assigned to the entry point at the start of the program, it's easier to use that register as the base register for data items referenced by instructions that require base-displacement addressability, as shown in Figure 541:

```
┌──────────────────────────────────┐
│Prog     Start                    │
│         J     Entry              │  ←— Jump over read/write data area
│         Using Prog,15            │  ←— Base register for read/write
│Data     DC    C'Message 1'       │      work area
│Packed   DS    PL5                │
│TenE24   DC    D'10E24'           │
│         ...   etc.               │
│SaveArea DS    18F                │
│         LTORG                    │
│ - - - - - - - - - - - - - - - - -│
│Entry    STM   14,12,12(13)       │  ←— Start of program instructions
│         LTR   1,1                │      that use only relative branch
│         JZ    NoArglist          │      instructions; all base—
│         CP    Packed,=P'0'       │      displacement references are
│         JE    NoPacked           │      are resolved with GR15
│         ...   rest of program    │
└──────────────────────────────────┘
```

Figure 541. Reorganizing a program to minimize base registers

As a general rule it's best to keep groups of instructions and groups of data in separate areas, for these reasons:

1. You'll very probably need fewer base registers, so you can use those registers for better purposes.
2. There is less likelihood of overwriting instructions (yes, it happens...).
3. Keeping read-only data such as constants separate from read-write work areas can help improve performance.
4. Modern processors buffer instructions and data from storage into different *caches*; if address references to one overlap with references to the other, the CPU can be slowed considerably.

> **Advice**
>
> To improve performance and minimize the number of base registers, keep instructions and data separated in the generated program.

Exercises

38.2.1.(2)+ Usually, we put USING statements after the BASR instructions that set the contents of the registers in the USINGs. In Figure 534 on page 799, why didn't we place the USING statement *after* the instruction whose address is in GR15?

38.2.2.(2)+ Explain the operation of the BASR 14,14 instruction in Figure 532 on page 798.

38.2.3.(2)+ In Figure 532 on page 798, what would happen if the **ShftRt** subroutine contains a USING *,14 statement at its entry point?

38.2.4.(1)+ In Figure 537 on page 800, suppose we had written the first instruction on the last line as

```
         L     5,=F'2147483647'
```

What results will be different between executing the L and LGFI instructions?

38.3. Separate Assemblies

Up to now we have treated all programs as self-contained. The entire program was translated in a single assembly; all instructions and declared data were processed by the Assembler as a single program. In practice it is often convenient (or even necessary) to assemble parts of programs separately, and link the parts later. While the parts could possibly be combined after debugging, and assembled in one large assembly, doing so is unnecessarily clumsy.

We now examine preparing programs as separately assembled segments, and how external symbols and address constants are used so that such routines can reference each other and their data areas.

We usually don't care exactly where a routine is in memory, so long as we can call it and have it produce the desired results. The **ShftRt** subroutine in Figure 534 on page 799 illustrates such a situation; the routine in Figure 533 on page 799 that calls it makes no assumptions about the actual location of the subroutine, only that the address constant at **AdShft** will contain the correct address.

Writing a routine once and using it often is more economical, so System z operating systems support various forms of program segmentation. The Linker combines separately assembled components into a complete program; we'll see how this works in Section 38.7.

The basic elements managed by the Linker are *control sections* and *external symbols*.

38.4. Control Sections

A control section is a segment of instructions or data (or both) that must be kept in one contiguous block in storage when the program is linked and executed. More precisely:

Control Section

A control section is the smallest indivisible contiguous segment of instructions and/or data which may be relocated independently of all other such segments, without affecting the logical operation of the program.

Correct execution of a program requires certain of its parts to maintain known and fixed relationships relative to one another. For example, the Assembler assigns Location Counter values within each assembly so it can correctly resolve the positions of implied addresses; if those positions are changed later, there's no guarantee the program will execute correctly

The following assembler instruction statements define a control section; for each, the name of the control section is provided in the name field entry of the statement. The first three identify what the *High Level Assembler Language Reference* calls *executable* control sections: they are part of the completed, linked program. Despite the "executable" name, they need not contain any executable instructions.

START START defines the beginning of a control section, and can set the initial value of the Location Counter to the operand, if specified. This initial value is rounded up to the next multiple of a doubleword (or other SECTALGN-specified) section alignment. For example,

```
MyProg   START 100                 Set initial LC for MyProg
```

would (by default) cause the Assembler to round the initial LC setting to 104 = X'000068', the next doubleword boundary. If the operand is omitted, zero is assumed. If a comment field is present when the operand is omitted, an isolated comma must separate it from the mnemonic, as in

```
MyProg   START ,                   Control section for MyProg
```

A START statement must be the first or only *executable* control section definition in a program. (It may be preceded by reference control sections, which are described below.)

CSECT A CSECT statement[249] defines a control section, and has no operand. Your program may define many control sections. If a comment field is present, an isolated comma should separate it from the mnemonic, as in

```
MySect7  CSECT ,                   Control section for MySect7
```

Unlike START, multiple CSECT statements are allowed in a single assembly.

RSECT The RSECT statement is exactly the same as a CSECT statement, with the additional property that the Assembler will check for possible modifications of areas within that control section. (This is sometimes called "reenterability" checking.)

The next three statements identify what the *High Level Assembler Language Reference* calls *reference* control sections: they generate no data or instructions. They can describe the organization of a data structure; only COM allocates storage for it.

COM Common control sections define work areas that can be shared by multiple control sections that may or may not be in the same assembly. They contain no assembly-time generated machine language instructions or data, but are part of the completed, linked program. COM statements have no operands.

DSECT A Dummy control section is a *template* or *mapping* of a memory area. A Dummy

[249] Occasionally, a control section is called a "CSect" (pronounced "See-Sect"). We usually know from context whether we are referring to a control section or to the assembler instruction statement with mnemonic CSECT.

control section generates no machine language object code.[250] Dummy sections are typically used to provide detailed symbolic descriptions of a data structure. (We'll see lots of examples in Chapter XI.)

DXD The DXD ("Define External Dummy") instruction defines one component of a DSect-like structure created by the Linker. (We'll describe DXD and related topics in Section 38.7.3.)

The Assembler maintains a separate Location Counter for each control section, and associates the control section name with the *first* byte of the control section. You can think of each control section as having its own "addressing space".

The only statements that may precede a control section definition are comments, PRINT controls, macro instruction definitions, and a few others of minor interest.

Be Careful!

If any statement that depends on or affects the LC, or which defines or declares a symbol, appears before any control section definition statement, an unnamed control section is automatically started with the LC initialized to zero. If a subsequent START statement appears, it is flagged as an error.

Each control section has its own relocation attribute, so you can't make implied-address references from one control section to another without additional USING statements. For example:

```
MyProg   Start 0                  My program
         Using *,15               Establish local addressability
         LA    1,SubData          Address of subroutine's data (Bad!)
         - - -
MySub    CSect ,                  My subroutine
         - - -
SubData  DC    F'1234'            Data in MySub
```

Figure 542. Incorrect implied reference to a different control section

The LA instruction will be diagnosed as an error, because there is no USING statement with the relocation attribute of **MySub**. To reference a field in a separate control section, we need something like this:

```
MyProg   Start 0                  My program
         Using *,15               Establish local addressability
         L     12,ASub            Get address of MySub
         Using MySub,12           Establish addressability
         LA    1,SubData          Address of subroutine's data (Good!)
         - - -
ASub     DC    A(MySub)           Address of MySub
         - - -
MySub    CSect ,                  My subroutine
         - - -
SubData  DC    F'1234'            Data in MySub
```

Figure 543. Correct implied reference to a different control section

After the second USING statement, the USING Table would contain two entries: one for register 15 with Relocation Attribute 01 and a second for register 12 with Relocation Attribute 02, illustrated in Figure 544 on page 805.

[250] As with "CSECT", a "DSECT" is often pronounced "Dee-Sect".

basereg	base location	RA
15	00000000	01
12	Loc of MySub	02

Figure 544. USING Table with two entries

Suppose a main program and our **ShftRt** subroutine are assembled together, but the subroutine is placed in a separate control section. Figure 545 shows how this can be done:

```
MyProg   Start 0                 Main program
         BASR  12,0              Establish base for main program
         Using *,12
         - - -
         L     0,Logic           Load GR0 and GR1
         L     1,Shift           ...with parameters
         L     15,AdShf          GR15 has subroutine address
         BASR  14,15             Link to subroutine
         ST    0,Result          Store result
         - - -
Logic    DS    F
Shift    DS    F
Result   DS    F
AdShf    DC    A(ShftRt)
         Drop  12                Don't want to use GR12 elsewhere
********************************
*        ShftRt Subroutine     *
********************************
ShftRt   CSect ,                 Separate control section for ShftRt
         Using *,15              Use caller's preset base register
         LTR   1,1               Test shift count
         JNM   ShftOK            Branch if not minus
         SR    1,1               Set shift count to zero
ShftOK   SRL   0,2(1)            Shift desired number of places
         BR    14                Return to caller
         END   MyProg            Begin execution in main program
```

Figure 545. Main program and subroutine in one assembly

The Assembler knows the location of **ShftRt**, and includes information in the object module so that the adcon named **AdShf** will contain the correct address at execution time. The two control sections (**MyProg** and **ShftRt**) are separately relocatable, so we should DROP the main program's base register before starting the code of the subroutine. Because none of the implied addresses in either control section refer to locations in the other, there is no need to provide addressability for more than one control section at a time.

To show how a COMMON control section can be shared between two executable control sections, we'll revise Figure 545 to put the items passed from **MyProg** to **ShftRt** in a common section named **ShfCom**. Figure 546 on page 806 shows how this can be done:

```
MyProg   Start 0                  Main program
         BASR  12,0               Establish base for main program
         Using *,12               Addressability for MyProg
         L     11,AdCom           Load address of ShfCom section
         Using ShfCom,11          Provide addressability
         - - -                    Calculate argument values
         ST    0,Logic            Store Logic argument in ShfCom
         ST    1,Shift            Store Shift argument in ShfCom
         L     15,AdShf           GR15 has subroutine address
         BASR  14,15              Link to subroutine
         ST    0,Result           Store result
         - - -
AdShf    DC    A(ShftRt)          Address of ShftRt subroutine
AdCom    DC    A(ShfCom)          Address of common section
         - - -
         DROP  12,11              Can't use GR12, GR11 elsewhere
ShfCom   COM   ,                  Declare common control section
Logic    DS    F                  Work areas in ShfCom
Shift    DS    F                  - - -
Result   DS    F                  - - -
********************************
*        ShftRt Subroutine     *
********************************
ShftRt   CSect ,                  Separate control section for ShftRt
         Using *,15               Use caller's preset base register
         L     1,ACom             Load address of common area
         Using ShfCom,1           Establish addressability
         L     0,Logic            Get argument to be shifted
         LT    1,Shift            Load and test shift count
         Drop  1                  We just modified that base register
         JNM   ShftOK             Branch if not minus
         SR    1,1                Set shift count to zero
ShftOK   SRL   0,2(1)             Shift desired number of places
         BR    14                 Return to caller
ACom     DC    A(ShfCom)          Address of common area
         Drop  15                 No longer needed for the subroutine
         END   MyProg             Begin execution in main program
```

Figure 546. Main program, subroutine, and common section in one assembly

38.4.1. Resuming Control Sections

You need not put all the statements belonging to a given control section together. That is, you can start a control section, then start another, and then resume the first one — and the Assembler will keep all the related pieces together in the generated object code. For example:

```
First    CSECT ,                  Start a control section
A        DC    F'101'
B        DC    F'103'
Second   CSECT ,                  Start another control section
W        DC    F'-107'
X        DC    F'-109'
First    CSECT ,                  Resume the 'First' control section
C        DC    F'113'
Second   CSECT ,                  Resume the 'Second' control section
Y        DC    F'-127'
```

Figure 547. Resuming control sections

In the control section named **First**, the fullword constants named A, B, and C will have locations X'0', X'4', and X'8' relative to the origin of the control section; similarly, the locations of the

three constants W, X, and Y in control section **Second** will have the same offsets relative to the origin of that control section.

As a more typical example to show how a CSECT statement can be used to resume a control section, we will rewrite Figure 545 on page 805 in the form shown in Figure 548, where the contributions to CSECT **MyProg** are defined in two separate segments.

```
MyProg   Start 0                  Start of main program
         BASR  12,0               Make a base
         Using *,12               Inform the assembler
         - - -
         L     0,Logic            Set up arguments in GR0
         L     1,Shift            ... and GR1
         L     15,AdShf           Subroutine address to GR15
         BASR  14,15              Link to subroutine
         Drop  12                 Can't use main base here
*
ShftRt   CSect ,                  Start of subroutine
         Using *,15               Addressability provided by GR15
         LTR   1,1                Test shift count
         JNM   ShftOK             Skip next instruction if not minus
         SR    1,1                Clear shift count
ShftOK   SRL   0,2(1)             Shift,
         BR    14                 And return
         Drop  15                 Can't use subroutine base here
*
MyProg   CSect ,                  Resume the main program CSECT
         Using MyProg+2,12        Re-establish Using information
         ST    0,Answer           Store answer (GR12 is base)
         - - -
Logic    DS    F                  Logical value to be shifted
SHIFT    DS    F                  Shift amount
Answer   DS    F                  Result
AdShr    DC    A(ShftRt)          ...and continue with program
```

Figure 548. Main program and subroutine in one assembly, multiple CSects

The **MyProg** CSect is defined in two pieces, and the necessary USING and DROP statements are inserted to ensure that implied addresses are correctly resolved. The **USING MyProg+2,12** statement is placed before the following ST instruction so that the implied address will be resolved with GR12 as the base register. Because this very simple subroutine has no implied addresses (and needs no local base register and no USING *,15) we could have omitted the "DROP 12" before the subroutine; but it's always a good programming practice to limit the range of each USING statement.

Advice

Limit the range of USING resolution to the minimum necessary; otherwise the Assembler may resolve implied addresses with base registers whose contents are no longer valid.

We can now see how an expression can have a complex relocation attribute. Suppose in the assembly in Figure 548 we wanted to calculate the offset of **ShftRt** relative to **MyProg** and wrote this statement:

```
         LA    2,ShftRt-MyProg
```

Because the two control sections are separately and independently relocatable (and the two symbols **ShftRt** and **MyProg** therefore have different relocation attributes), there is no way the Assembler can assign a base register and an absolute displacement to the instruction.

38.4.2. Literals in Multi-Section Assemblies (*)

Be careful when you use literals in programs with more than one control section. Literals are symbols, so they can have all the addressability problems of ordinary symbols. In particular, the Assembler could place the literals in a part of the program that is not addressable!

If any literals are left in the Assembler's literal table at the end of the assembly, they are placed at the end of the *first* executable control section defined in the program. This may or may not be what you intended. Either (1) don't use literals in a multi-CSect assembly, or (2) use LTORG statements at the end of each control section, so that the literals used in each control section will be defined within it.

For example:

```
Section1 Start 0
         - - -
         L     2,=F'1'
         - - -
Section2 CSect ,
         - - -
         L     7,=F'43'
         - - -
         End
```

Because there is no LTORG in control section **Section2**, both literals will be placed in a literal pool at the end of **Section1**. This probably means that references in **Section2** to literals will generate addressability errors.

38.4.3. Location Counter Discontinuities (*)

Because the Assembler uses multiple, separate Location Counters for control sections, they can also be used to manage *discontinuities* in the Location Counter. Suppose we write these statements, and the Assembler knows the value of the Location Counter when it processes the first statement:

```
Loc     Statement
000046  A        DS    XL(N)              N is not defined yet
?       B        DS    F                  Aligned Fullword
?       C        DS    CL(3*N)            Another discontinuity
?       D        DS    F                  Aligned Fullword
000005  N        Equ   5                  Finally, define N
```

Figure 549. Statements with Location Counter discontinuities

Because the value of the symbol **N** is not yet known, the location and alignment of the symbol **B** can't be assigned. So, the Assembler assigns a temporary separate Location Counter to those two statements and saves information about this “group” of two statements until the value of **N** is known. Similarly, the statements named **C** and **D** represent another discontinuity group with their own temporary Location Counter.

When the END statement is reached, the unresolved discontinuity-group segments are analyzed. If they are resolvable (as in this case), the temporary Location Counter values are “stitched” into their proper places among the preceding and following segments. Thus, if the value of the Location Counter is X'000046' when the first statement in Figure 549 is processed, the value of the symbol **B** will be X'00004C'.

If any group cannot be resolved, the Assembler issues a diagnostic and ignores the faulty statement:

```
G        DS    XL(K)              K is not defined yet
 ** ASMA080E Statement is unresolvable
H        DS    F                  Aligned Fullword
K        Equ   *-G                Now, G is defined (?)
 ** ASMA044E Undefined symbol - G
```

Because the symbol **G** couldn't be assigned a value (that is, be defined) until the value of **K** was known, the alignment of **H** was unknown and therefore resolving the value of **K** could not be completed.

38.4.4. Section Alignment (*)

The first byte of an executable or COMMON control section is always placed in memory on an aligned boundary. By default, all control sections are aligned on a doubleword boundary, but occasionally you may require a more stringent alignment. If you specify the SECTALGN option, the Assembler will align all control sections on the boundary you specify.

For example, if you specify the SECTALGN(16) option, all control sections will be aligned on a quadword boundary.[251]

When your program is linked with other control sections, this may waste a few bytes if the preceding control section does not end just before such a boundary. However, the Linker and Program Loader will respect your alignment request and load the executing program on the requested boundary. This means that boundary alignments *within* a control section (due to CNOP, ORG, DC, or DS statements) are preserved relative to the start of the control section.

In some cases, you can adjust the LC within a CSECT to a stricter boundary than the default, and then rely on Linker controls to request that the CSECT be aligned on (for example) a page boundary, thus guaranteeing that the stricter internal alignment will be honored when the program is linked. For example, suppose you want to force a CSECT to be an exact number of 4K “pages” long, by adding enough bytes at the end to force its length to be a multiple of 4096:

```
Prog     CSECT ,                 Start a control section
         - - -                   Add lots of statements
         LtOrg ,                 Insert the literals now
*        Round the LC to a page boundary
         DC    ((((*-Prog)+4095)/4096)*4096-(*-Prog))X'00'
ProgLen  Equ   *-Prog            Will now be a multiple of X'1000'
         End
```

Figure 550. Technique for rounding the length of a CSECT

38.4.5. Threaded Location Counters (*)

Location Counter values in multi-CSECT assemblies are occasionally puzzling: the leftmost column of the Assembler's listing shows the value of the LC assigned to each statement. When a single control section is being assembled, these LC values increase predictably as succeeding bytes are assembled.

In a multi-CSect assembly, the values of the LC may appear to vary as the control section changes. Even though the statements belonging to different CSects are interleaved, the LC values look as though the statements had been rearranged within each control section, as noted in Section 38.4.1 above.

Remember that the Assembler maintains a *separate* Location Counter for each control section in an assembly. Except for the first CSECT (whose LC may be initialized to a nonzero value on a START statement), these LC values are set to zero whenever a new control section is encountered. Then, whenever a CSECT statement indicates that a different control section is to be begun or resumed, the Assembler uses a new LC to continue the assembly.

After the first pass of the assembly is complete, the Assembler processes its collection of Location Counters. First, the length of each CSECT is determined; this is possible because the Assembler knows both the current and the maximum value of the LC for each control section. Then, beginning with the first CSECT (whose initial LC value is known), the Assembler adds the initial LC value for each control section to its length, rounds the sum up to the next alignment boundary,

[251] Not every system Linker supports quadword alignment, so you should try a simple test. Section alignments stricter than quadword require specifying the GOFF object-file option, or providing control statements to the Linker.

and assigns the resulting value as the initial LC value of the *next* CSECT. In this way, the control sections appear to have been "unscrambled" and assembled end-to-end.

During the second assembly pass, the Assembler uses these adjusted LC values to compute the values of expressions involving symbols and Location Counter references. The relocation attribute of the symbol tells which CSECT the symbol belongs in, and therefore what LC value should be added to the symbol's value found in the symbol table.

Adjusting the LC values so that each control section starts at the next aligned boundary following the end of the previous one is called *threading* the location counters. With no threading, each control section starts at location zero.

Why does the Assembler perform threading? In the earliest days of System/360, programs were not organized into complex load modules by the Linkage Editor, but were loaded directly into memory from object modules. Because the address where loading began was often known in advance, that address could be assigned as the initial LC value in a START statement. Thus, the LC values printed on the Assembler's listing corresponded exactly to the addresses in memory occupied by the assembled and loaded program. This made debugging simpler, because an interruption address could be immediately identified with the offending instruction in the listing.

Threading was intended as a convenience for you. We will see in discussing program linking that the LC values must be "un-threaded" by the Linker so it can correctly relocate the program. Whether threading is a help or hindrance is a matter of personal preference.

Sometimes it's easier to debug programs if each control section starts at location zero (except possibly the first if its LC value is set by a START statement). If you specify the NOTHREAD option, the Assembler will not "thread" LC values for each new control section.

38.4.6. The "Location Counter" Instruction LOCTR (*)

As discussed in Section 38.4.2 on on page 808, the Assembler can create separate Location Counters to handle discontinuities. The Assembler lets you create your own Location Counter discontinuities in Assembler Language programs with the LOCTR[252] instruction.

LOCTR can help you group segments of your program so that the order of statements in each control section of the *source* program need not be the same as the order of the generated instructions and data in the *object* program, as was done in all our previous examples.

The LOCTR instruction starts (or continues) a separate group of statements having its own Location Counter. When the end of the source program is reached, the Assembler collects the portions of each LOCTR group in the order they were declared into a single group, and assigns a sequential Location Counter value to each item within each named group. Figure 551 on page 811 illustrates this process:

[252] The LOCTR instruction describes a Location Counter group, so its pronunciation would logically be LOKE-ter. But the people who invented it called it LOCK-ter. Your preference?

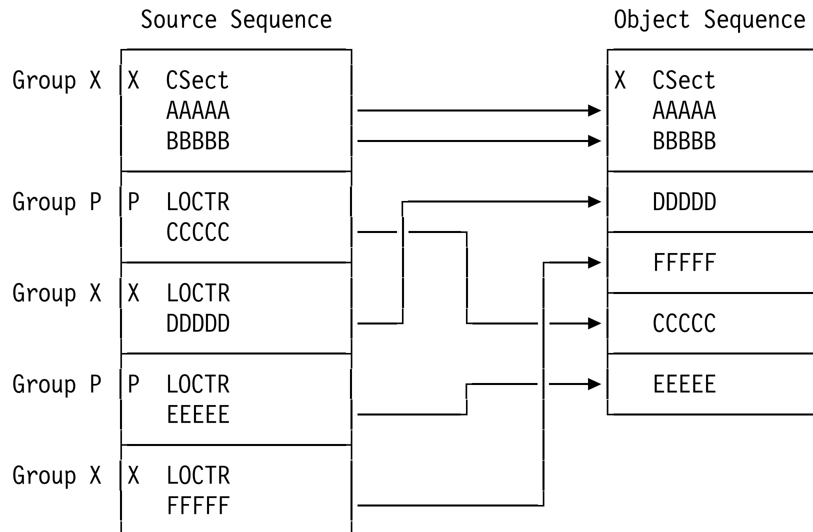


Figure 551. Rearrangement of source groups by LOCTR

The three portions of LOCTR group **X** appear first in the generated object code, followed by the three portions of LOCTR group **P**. All the statements of the two LOCTR groups are part of the control section named **X**.

Note that all the statements are in control section **X** and that we have simply reordered the “normal“ connection between source and object components of that control section. Conversely, multiple CSECT statements don't change the ordering relation between statement sequence and the object code generated for each control section.

Figure 551 illustrates important properties of LOCTR groups:

1. A statement starting a control section (like CSECT) starts a LOCTR group with the name of the control section. Thus, each control section is its own LOCTR group!
2. The name of a LOCTR group appears in the name field of the LOCTR assembler instruction statement. Statements following the LOCTR belong to that group. Thus, the two groups in the figure are named X and P.
3. The *object* sequence of source statements is in the order of the first appearance of the group's declaration on (1) a statement that initiates a control section (START, CSECT, DSECT, RSECT, COM) or (2) a LOCTR statement.
4. A LOCTR group can be interrupted by other LOCTR groups, and then resumed later in the source program. The same capability for control sections was described in Section 38.4.1..

These simple examples illustrate the effects of LOCTR:

1. This example starts LOCTR group **ALoc** and does not resume the LOCTR group of control section **A**.

```
A      CSect ,           Define LOCTR group A
       DC    X'00'     Location = X'000000'
ALoc   LOCTR ,           Define LOCTR group ALoc
       DC    X'01'     Location = X'000001'
       End
```

Figure 552. Simple example of LOCTR (1)

2. This example resumes the LOCTR of control section **A** after the **ALoc** LOCTR was defined.

```
A      CSect ,           Define LOCTR group A
ALoc   LOCTR ,           Define LOCTR group ALoc
       DC    X'01'     Location = X'000001'
A      LOCTR ,           Resume LOCTR group A
       DC    X'00'     Location = X'000000'
       End
```

Figure 553. Simple example of LOCTR (2)

3. In this example, both LOCTR groups are first defined and then resumed.

```
A      CSect ,           Define LOCTR group A
ALoc   LOCTR ,           Define LOCTR group ALoc
A      LOCTR ,           Resume LOCTR group A
       DC    X'00'     Location = X'000000'
ALoc   LOCTR ,           Resume LOCTR group ALoc
       DC    X'01'     Location = X'000001'
       End
```

Figure 554. Simple example of LOCTR (3)

While many programs don't need the capability provided by LOCTR, it can improve readability and understandability by keeping related portions of the source program close together, while also keeping them separated in the object program. For example, though the simple program fragment in Figure 539 on page 801 may be more understandable in that form, we may eventually want the generated instructions and data to appear as in Figure 540 on page 801.[253] Figure 555 uses LOCTR groups named **Code** and **Data** to do this:

```
         Using Data,12
         - - -
Code     LOCTR ,                Start LOCTR group for instructions
         L     2,X
         A     2,Y
         ST    2,Z
         - - -
         JP    Positive
         - - -
Data     LOCTR ,                Start LOCTR group for data
X        DC    F'something'
Y        DC    F'something_else'
Z        DS    F                Some sum
Code     LOCTR ,                Resume LOCTR group for instructions
         - - -
Positive L     0,PlusCode       Indicate positive sum
         - - -
Data     LOCTR ,                Resume LOCTR group for data
PlusCode DC    F'some number'
```

Figure 555. A program fragment using LOCTR for reorganization

The object code generated by this program fragment would be the same as that generated in Figure 540 on page 801.

Organizing a program using LOCTR to keep instructions and data separated is sketched in Figure 556 on page 813:

[253] Closely mixing instructions and data as in Figure 539 on page 801 can have significant negative effects on program performance.

```
MyProg   CSect ,                 Start of My Program
         J     Start             Relative branch to instruction LOCTR
Consts   LOCTR ,                 Start a LOCTR group for constants
Msg1L    DC    AL1(L'Message1)   Length of message
Message1 DC    C'This is a message string'
Data     LOCTR ,                 Start a LOCTR group for writable data
Number   DS    CL12              Binary number converted to dec. chars
Code     LOCTR ,                 Start of LOCTR group for instructions
Start    STM   14,12,12(13)      Save registers in caller's save area
         LR    12,15             GR12 is base register for Consts/Data
         Using MyProg,12         Addressability for Constants and Data
         L     4,=F'-23456'      Load value from literal pool
         - - -
         OI    Number+L'Number-1,X'F0' Make last converted digit EBCDIC
         - - -
         LB    1,Msg1L           Get length of Message 1
         - - -
Consts   LOCTR ,                 Return to constants LOCTR group
         LTORG ,                 Put literals in Consts LOCTR group
         End
```

Figure 556. Organizing a program to minimize addressability problems

After assembly, the program in Figure 556 appears *as though* it had been written this way:

```
MyProg   CSect ,                 Start of My Program
         J     Start             Relative branch to instruction LOCTR
Msg1L    DC    AL1(L'Message1)   Length of message
Message1 DC    C'This is a message string'
         LTORG ,                 Emit literals in Consts LOCTR group
Number   DS    CL12              Binary number converted to dec. chars
Start    STM   14,12,12(13)      Save registers in caller's save area
         LR    12,15             GR12 is base register for Consts/Data
         Using MyProg,12         Addressability for Constants and Data
         L     4,=F'-23456'      Load value from literal pool
         - - -
         OI    Number+L'Number-1,X'F0' Make last converted digit EBCDIC
         - - -
         LB    1,Msg1L           Get length of Message 1
         - - -
         End
```

Figure 557. Organizing a program to minimize addressability problems

Aside from the reordered statements, note these differences between Figure 556 and Figure 557:

- All the constants and read-write areas are close to the origin of the control section, so will probably be within the range of addressability of the single USING statement.
- The literal in the program follows the LTORG statement; but because LTORG appeared at the *end* of the source program the literal will be generated with the other constants near the *start* of the object program.
- Branch instructions in the **Code** LOCTR group can now be changed to relative-immediate instructions, so no additional addressability is needed within that LOCTR group. In larger programs, this can free up previous base registers for more productive use.

  Thus, it may be possible that all based references to the **Consts** and **Data** LOCTR groups can be addressed with a single base register. And, if you use relative-addressing branch instructions in the **Code** LOCTR group, then you shouldn't need a base register to address any instructions.

LOCTR also facilitates correctly defining the target of an EX instruction, since its definition will be close to the EX that references it. This helps ensure that the current USING instructions will be honored in both the EX and the target instruction.

```
Program  Loctr ,                 Group for mainline instructions
         - - -
         EX    1,Target          Reference the target instruction
Constant Loctr ,                 Put the target with constants
Target   <the target instruction>
Program  Loctr ,                 Resume the mainline instructions
```

Some unexpected behaviors may arise when you mix control section and LOCTR statements.

1. This example uses both CSECT and LOCTR, where we define LOCTR groups in more than one control section:

```
A       CSect ,           Define LOCTR group A
        DC    X'00'    Location = X'000000'
B       CSect ,           Define LOCTR group B
        DC    X'02'    Location = X'000008'
A       LOCTR ,           Resume LOCTR group A
        DC    X'01'    Location = X'000001'
B       LOCTR ,           Resume LOCTR group B
        DC    X'03'    Location = X'000009'
        End
```

Figure 558. Simple example of LOCTR (4)

While control section **B** is active, the following "A LOCTR" statement resumes control section **A**. Resuming a LOCTR group can cause a change of control section.

2. This example inserts a second "A CSECT" statement, with "interesting" results:

```
A       CSect ,           Define LOCTR group A
        DC    X'00'    Location = X'000000'
ALoc    LOCTR ,           Define LOCTR group ALoc
        DC    X'01'    Location = X'000001'
A       CSECT ,           Resume LOCTR group A? ◄— Note!
        DC    X'02'    Location = X'000002'   !
        End
```

Figure 559. Example of unexpected LOCTR behavior (1)

Resuming CSECT **A** does *not* resume its Location Counter! You might have expected that the last DC statement would follow the first. But, as the Location values of the constants indicate, *resuming a control section resumes the most recently active LOCTR group in that control section*, not the LOCTR group associated with the control section name.

3. Here's another example that illustrates this behavior:

```
A      CSect ,          Define LOCTR group A
       DC    X'00'     Location = X'000000'
ALoc   LOCTR ,          Define LOCTR group ALoc
       DC    X'01'     Location = X'000001'
B      CSect ,          Define LOCTR group B
       DC    X'02'     Location = X'000008'
BLoc   LOCTR ,          Define LOCTR group BLoc
       DC    X'03'     Location = X'000009'
A      CSect ,          Try to resume LOCTR group A
       DC    X'04'     Location = X'000002' (not X'000001'!)
B      CSect ,          Try to resume LOCTR group B
       DC    X'05'     Location = X'00000A' (not X'000009'!)
       End
```

Figure 560. Example of unexpected LOCTR behavior (2)

You might have expected both CSECT statements to resume their LOCTR groups; they instead resume the most recently active LOCTR groups within their sections. That's why the constant X'04' does not follow X'00', but X'04' instead follows X'01'. The constants in control section **B** behave similarly.

We can summarize these rules for using LOCTR statements and statements naming control sections.

1. A LOCTR group does not start a separate or different control section unless the LOCTR statement is the first LC-dependent statement in the program (it does depend on the Location Counter), in which case a Private Code control section is initiated by the Assembler. (Starting a program with LOCTR isn't a good idea!)
2. A LOCTR group is part of the control section in which the group is first declared. The portions of a LOCTR group belong to the control section being assembled when its first LOCTR statement appears. That is, the statements in a LOCTR group have the same relocatability attribute as the owning control section in which they appear.
3. Resuming a control section resumes the most recently processed LOCTR group in that control section. If no LOCTR group has been defined in the control section, the control section is resumed normally. That is, you can use a LOCTR group name to resume a control section, but a control section name cannot safely be used to resume a LOCTR group. Instead, it resumes the *most recently used* LOCTR group in that control section.
4. To resume a control section in which LOCTR groups have been defined, issue a LOCTR statement with the name of the control section.
5. A change of LOCTR group may also change the current control section.
6. ORG in a LOCTR group pertains only to that group, and cannot be used to switch the Location Counter to a different control section.

Thus, the Assembler knows how to manage groups of statements with their own temporary Location Counters; the LOCTR statement lets you declare *and* determine the order of your own groups of statements.

Advice

Don't use a LOCTR name as a branch target, because you may not know which of many possible occurrences of the name is the actual target.

Exercises

38.4.1.(2)+ In this program fragment, where will the literals =F'1' be generated?

```
AAA      CSect ,
         - - -
         L     3,=F'1'
         - - -
         LTORG
BBB      CSect ,
         - - -
         S     8,=F'1'
         - - -
         End
```

38.4.2.(2) In this program fragment, where will the literals =F'1' be generated?

```
CCC      CSect ,
         - - -
         L     3,=F'1'
         - - -
         LTORG
DDD      CSect ,
         - - -
         S     8,=F'1'
         - - -
         LTORG
         End
```

38.4.3.(1) In Figure 545 on page 805, why is a DROP 15 statement not needed at the end of the **ShftRt** subroutine?

38.4.4.(2)+ In Figure 545 on page 805, why is the USING statement in the **ShftRt** control section not needed? Is there any reason why it should be retained?

38.4.5.(2) In Figure 548 on page 807, why is it a good idea to leave the USING *,15 statement at the start of the **ShftRt** subroutine, even though it's not required?

38.4.6.(1)+ In Figure 549 on page 808, determine the values of all symbols if symbol **A** has value X'001240'. Now, do Exercise 38.4.7.

38.4.7.(2)+ In Figure 549 on page 808, why is the value of the symbol **B** not the value of the symbol **A** plus five, or X'00004B'?

38.4.8.(2)+ First, make a complete and correct Assembler Language program using the instructions in Figure 546 on page 806. Then, assemble it twice: first with the THREAD option, and again with the NOTHREAD option. Study the Assembler's Location Counter values for each control section.

38.4.9.(2)+ What Location Counter values are assigned to the symbols naming DS statements in this program?

```
Prog     Start X'2000'
A        DS    X
NewGroup LOCTR ,
B        DS    H
Prog     LOCTR ,
C        DS    F
NewGroup LOCTR ,
D        DS    F
         End
```

38.4.10.(3)+ In Figure 550 on page 809, explain how the expressions in the DC statement generate the desired alignment.

38.4.11.(2) In this little program, in which control sections will each constant reside? Assuming normal doubleword section alignment, what will be their locations if the Location Counter values are “threaded”?

```
A        Start 0
         DC    X'1'
B        CSect ,
         DC    X'2'
C        CSect ,
         DC    X'3'
B        CSect ,
         DC    X'4'
C        CSect ,
         DC    X'5'
A        CSect ,
         DC    X'6'
C        CSect ,
         DC    X'7'
         End
```

38.4.12.(2)+ What Location Counter values are assigned to the symbols naming DS statements in this program?

```
Loc      CSect ,
H        DS    X
ALoc     LOCTR ,
A        DS    X
BLoc     LOCTR ,
B        DS    X
Loc      LOCTR ,
C        DS    X
ALoc     LOCTR ,
D        DS    X
BLoc     LOCTR ,
E        DS    X
Loc      LOCTR ,
F        DS    X
         End
```

38.4.13.(3)+ If you assemble this program, what will be the Location Counter values assigned to each symbol?

```
SectA    CSect ,
H        DS    X
ALoc     LOCTR ,
B        DS    X
SectB    CSect ,
A        DS    X
BLoc     LOCTR ,
C        DS    X
ALoc     LOCTR ,
D        DS    X
BLoc     LOCTR ,
E        DS    X
SectA    CSect ,
F        DS    X
BLoc     LOCTR ,
G        DS    X
SectB    LOCTR ,
N        DS    H
```

38.4.14.(2) Section 38.4.2 noted that literals not previously generated by LTORG statements are placed at the end of the first executable control section when the END statement is processed. How do you think the Assembler does that?

38.5. External Symbols

Usually, none of the symbols you used to write your program are available after the completion of its assembly, except for the program name. To communicate among routines assembled or compiled at different times, we need the additional information provided by *external symbols.*.

An external symbol may be one of the following five types, and the statements used to declare them are shown in parentheses.

1. Control section name (CSECT, RSECT, START)

   A *control section name* is the name of a block of instructions or data. The name identifies the control section; it is used by the Linker to build an executable load module. Names of "executable" control sections — defined by START, CSECT, and RSECT instructions — are always external symbols.

2. Common section definition (COM)

   A common section is a reference control section that generates no assembly-time machine language instructions or data. Typically, COM sections are shared work areas among executing CSECTs in a loaded program. (That is, they are "commonly" available to multiple code CSECTs.) They provide an easy way to reference large data aggregates without passing their addresses as subroutine arguments.

3. External reference (EXTRN, WXTRN)

   An *external reference* is a symbol that is not assigned a value at assembly time. It is usually the name of an instruction or data area that will be supplied and resolved during program linking, often from the "Library of Object and Load Modules" illustrated in Figure 24 on page 73. EXTRN and WXTRN statements are needed *only* for symbols not defined in the assembly in which references to those symbols are needed.

   External references are typically used in address constants that, after program linking is complete, will contain addresses of *other* program segments.

   V-type address constant operands are automatically declared EXTRN; they will be described in Section 38.5.2.

4. Entry name (ENTRY)

   An *entry name* is associated with a particular location *within* a control section. For example, library routines calculating the trigonometric functions SIN and COS use nearly identical algorithms, so a single control section may have separate entry names for SIN and COS.

5. Dummy external section (DXD, and DSECT in special cases)

   Dummy external sections are sometimes called "PseudoRegisters". They have specialized uses that we'll describe briefly in Section 38.7.3. Their names may match other external symbol names without conflict.

The Assembler maintains a special symbol table called the *External Symbol Dictionary* (ESD) for external symbol information.[254] Each item in the External Symbol Dictionary (other than ENTRY names) is assigned an "ESD ID" number, or "ESDID". (An ENTRY name is given an "Owning ID", the ESDID of the Section to which it belongs.)

[254] Some of this information may also appear in the Symbol Table for "ordinary" (internal) symbols.

38.5.1. EXTRN and WXTRN Statements

Suppose our **ShftRt** subroutine has already been assembled as a separate program, to be combined by the Linker with our calling program. The symbol **ShftRt** will be defined somewhere outside our calling program — to the caller, it is an *external* symbol. Thus, the Assembler must provide information to the Linker indicating that we want to reference that symbol. This is done with the EXTRN statement.

For example, the subroutine call in Figure 533 on page 799 can be modified to make **ShftRt** an external symbol.

```
         EXTRN ShftRt            Declare 'ShftRt' to be external
         - - -
         L     0,Logic           Datum to be shifted, in GR0
         L     1,Shift           Shift amount, in GR1
         L     15,AdShft         Subroutine address, in GR15
         BASR  14,15             Link to subroutine
         ST    0,Result          Store shifted result
         - - -
Logic    DS    F                 Space for arguments,
Shift    DS    F                 ...
Result   DS    F                 And result
AdShft   DC    A(ShftRt)         Subroutine address
```

Figure 561. Calling ShftRt as an external routine

When we compare the examples in Figure 533 on page 799 and Figure 561 above, the only difference is the presence of the EXTRN statement. The true address of **ShftRt** in the address constant named **AdShft** will be created by the Program Loader when the program is loaded into memory.

We can write the **ShftRt** routine as a separately assembled routine. First, we create a complete program in which the subroutine is defined as a control section named **ShftRt**. Because a control section name is already an external name, no EXTRN declaration is needed.

```
ShftRt   Start 0                 Define control section name
         Using *,15              GR15 preset by caller
         LTR   1,1               Test shift count
         JNM   ShftOK            Branch if not minus
         SR    1,1               Set shift to zero
ShftOK   SRL   0,2(1)            Perform the shifts
         BR    14                Return to caller
         END
```

Figure 562. ShftRt subroutine as a separate assembly

This assembly generates an object module with the single **ShftRt** control section, 14 bytes long. The only symbol in the External Symbol Dictionary in the object module will be the control section name. The “program name” is not an operand of the END statement, because we are not writing a main program that should be executed starting at **ShftRt**.

You can also use external symbols with relative branch instructions.[255] For example, you can use references like these:

[255] While the Assembler supports relative-immediate external references, you should verify that the Linker(s) in your operating system environment also supports them.

```
         EXTRN A,B
         - - -
         BRAS  14,A              Call nearby external routine A
         - - -
         BRASL 14,B              Call distant external routine B
```

Figure 563. External references using relative branch instructions

The Assembler encodes the 2-byte (for BRAS) and 4-byte (for BRASL) relative references in the object module. BRAS would typically be used to refer to external routines in a linked program not longer than (say) 50K bytes, while BRASL can be used for very large programs or programs in which different parts are widely separated.

Note: If you use instructions like those in Figure 563, GR15 will *not* contain the entry point address!

The EXTRN and WXTRN assembler instruction statements have no name-field entry. The operand field is a symbol or list of symbols separated by commas. Each symbol is entered in the Assembler's External Symbol Dictionary, and given its own relocation attribute because the Assembler must assume it is independently relocatable.

The only difference between EXTRN and WXTRN symbols occurs during linking: if all the input object and load modules provided to the Linker do not have a definition of the symbol, EXTRN causes a library search for a definition, while WXTRN does not.

You can use WXTRN to declare and test for external symbols that are not linked with the completed program. The example in Figure 564 shows how you can do this:

```
         WXTRN Sub1              Weak reference for Sub1
         EXTRN Sub2              Strong reference for Sub2
         - - -
         LT    15,ASub1          Test if Sub1 was linked
         JZ    CallSub2          Branch if it's not available
         BASR  14,15             It's available, call it
         - - -
CallSub2 L     15,ASub2          Get the address of Sub2
         BASR  14,15             ... and call it
         - - -
ASub1    DC    A(Sub1)           Address of Sub1, if linked
ASub2    DC    A(Sub2)           Address of Sub2
```

Figure 564. Using WXTRN to test whether a routine was linked

An external name declared in a WXTRN statement will not cause any Linker diagnostics if the name cannot be linked. Then, you can test the address constant referencing the name for zero to determine whether or not it is present in the linked program.

A symbol cannot appear both as an operand of an EXTRN or WXTRN statement *and also* as the name-field symbol in a statement which would cause the Assembler to assign a value to it, because this combination would be an attempt to declare that the symbol both is, and is not, defined external to this program.

```
         EXTRN NotGood
NotGood  DC    C'Not So Good'    Internal symbol
** ASMA043E Previously defined symbol - NotGood
```

38.5.2. V-Type Address Constants

Calling an external subroutine can be simplified if you use V-type address constants. The presence of a symbol in a V-type adcon *automatically* and *implicitly* declares the symbol as external. We could rewrite Figure 561 on page 819 as shown in Figure 565 on page 821. The differences are the omission of the EXTRN statement, and using a V-type adcon in place of an A-type adcon.

```
*        EXTRN ShftRt               Not needed, we're using a V-con
         - - -
         L     0,Logic              Datum to be shifted, in GR0
         L     1,Shift              Shift amount, in GR1
         L     15,AdShft            Subroutine address, in GR15
         BASR  14,15                Link to subroutine
         ST    0,Result             Store shifted result
         - - -
Logic    DS    F                    Space for arguments,
Shift    DS    F                    ...
Result   DS    F                    And result
AdShft   DC    V(ShftRt)            Subroutine address in a V-con
```

Figure 565. Calling ShftRt as an external routine

Before converting programs using EXTRN declarations and A-type address constants to use V-type adcons, take note of these considerations:

1. The operand of a V-type address constant may be a symbol only.
2. In the object module, an additional flag is attached to the external symbols that appear in V-type address constants. This flag means that the symbol is the name of a routine or entry point to which control may be passed, as well as possibly the name of a data area.

   There is no distinction between a V-type adcon and an A-type plus EXTRN *unless* the routine to which control may be transferred might use an assisted linkage.[256] In this special case, the branch will actually go to a linkage-assist routine that loads the target routine into memory if necessary and then branches to it.

   This is why an expression cannot be allowed in a V-type adcon: the branch may go *not* to the target routine but to a linkage-assist routine, so no added factors can be allowed in the address. Assisted linkage is (fortunately) usually invisible.
3. The symbol in a V-type constant may also be explicitly declared in an EXTRN or WXTRN statement.

Some useful programming conventions for V-type constants and external symbols are:

1. All external symbols should be explicitly declared in EXTRN statements, even though an implicit declaration could have been used.
2. All external branch addresses should be referenced through V-type constants. If you know that calling a routine will never use an assisted linkage, you can replace V-type constants by A-type constants (but retain the EXTRN declarations you've done already).
3. Data areas should *never* be addressed using V-type address constants: if assisted linkages are used at some later time, the address in the V-con might not be the address of the data area! And, if you *store* data at the address referenced by the constant, you might be destroying important linkage information.

38.5.3 ENTRY Statement

In addition to specifying **ShftRt** as a control section name, you can use an ENTRY statement to identify **ShftRt** as the name of a location in a control section with a different name. As with EXTRN, symbols identified as entry points are written as operands of an ENTRY statement, separated by commas.

ENTRY symbols *must* be defined in the program in which the ENTRY statement occurs. Otherwise, there would be no position in the control section that could be identified as the desired entry point.

[256] This was particularly true for load modules formed into overlay structures, which are little used today; but other forms of assisted linkage exist, such as Dynamic Link Libraries (DLLs).

Suppose we rewrite the **ShftRt** subroutine as part of a control section named **SubShf2**. (We use the argument-passing convention of Figure 489 on page 766 in which the result address is also provided as an argument, and all registers are preserved.)

```
SubShf2  Start 0                    Set control section name and LC
         ENTRY ShftRt               Identify entry point
         Using *,15                 Assume standard linkage
ShftRt   STM   0,4,Save             Save GR0 through GR4
         LM    2,4,0(1)             Get argument and result addresses
         L     0,0(0,2)             Logical datum to GR0
         L     1,0(0,3)             Shift amount in GR1
         SRL   0,2                  Perform initial shift
         LTR   1,1                  Test additional shift count
         JNP   Finish               Branch if not positive
         SRL   0,0(1)               Perform rest of shift
Finish   ST    0,0(0,4)             Store result at given location
         LM    0,4,Save             Restore all registers we used
         BR    14                   And return to caller via r14
Save     DS    5F                   GR0-GR4 register save area
         END
```

Figure 566. ShftRt subroutine in a different CSect

The External Symbol Dictionary for this assembly contains the CSect name **SubShf2** and the entry name **ShftRt**, and information to indicate the type of each symbol.

To illustrate a typical use of the ENTRY statement, suppose a **Main** routine calls a subroutine **Subr**. The data for the subroutine is in the main program, named **Data**. Rather than passing the address of the data area to the subroutine, the main program identifies it with an ENTRY statement.

```
Main     Start 0                    Main program
         ENTRY Data                 Identify entry point of 'Data'
         BASR  12,0                 Establish base register
         Using *,12                 Assembler resolves displacements
         - - -                      Compute something in data area
         L     15,ASubr             Get subroutine address
         BASR  14,15                Branch to subroutine
         - - -                      Do something with the results
ASubr    DC    V(Subr)              External subroutine address
Data     DS    200F                 Data area
         END   Main                 Start execution in main program
```

Figure 567. Main program with ENTRY for data

The separately assembled subroutine **Subr** could be written as in Figure 568:

```
Subr     Start 0                    Define control section 'Subr'
         EXTRN Data                 Mark 'Data' as external
         Using *,15                 Standard linkage assumed
         STM   0,6,Save             Assume we need only GR0-GR6
         L     6,AData              Get address of data area
         L     0,0(0,6)             First data word to GR0
         - - -                      Work with all the data
         LM    0,6,Save             Restore GR0-GR6
         BR    14                   Return to master
AData    DC    A(Data)              Address of external data area
Save     DS    7F                   Save area for registers GR0-GR6
         END
```

Figure 568. Subroutine using EXTRN to reference data

Each of these programs refers to a name defined in the other by using an EXTRN declaration and an appropriate address constant.

We can use address constants for addressing data in many ways. For example, suppose the data area accessed by the subroutine contains several sub-areas that begin at **DATA**, **DATA+60**, and at **DATA+453**. We also suppose the subroutine refers to these sub-areas often enough that it keeps their addresses in registers. Because A-type address constants may contain expressions involving external symbols, we can define three adcons to point to the parts of the data area.

```
Subr     Start 0               Define the CSect name
         EXTRN Data            Indicate external symbol
         Using *,15            Caller will preset GR15
Subr     STM   0,6,SAVE        Save GR0-GR6
         LM    4,6,ADatas      C(GR4) = A(start of data area),
*                              GR5 points to second data area,
*                              And GR6 points to the third
         - - -                 ... work with the data
         LM    0,6,Save        Restore registers we used
         BR    14              Return to caller
ADatas   DC    A(Data)         Address of base of data area
         DC    A(Data+60)      Address of next sub-area
         DC    A(Data+453)     Address of last data sub-area
Save     DS    7F              Register save area
         END
```

Figure 569. Subroutine using EXTRN and adcons to reference data

The usage illustrated in Figure 569 cannot be “simplified” by using V-type constants such as V(DATA+60).

Because multiple operands are allowed in address constants, we could also have written

```
ADatas   DC    A(Data,Data+60,Data+453)
```

or we could have used a literal, and written the LM instruction as

```
         LM    4,6,=A(Data,Data+60,Data+453)
```

Adcons containing expressions involving external symbols must be used with care. If the referenced routine reorders the statements defining the data, the offsets encoded in the adcons will probably be incorrect. It's better to identify such entry points by name, not by offset from another symbol.

Sometimes we must write two or more subroutines that do almost the same thing, but only minor variations are needed in each. We can combine the routines into a single CSect and use as much common coding as possible.

Suppose we need not only the **ShftRt** subroutine, but also one with identical parameters named **ShfLft**, which does a logical *left* shift instead. First, we will sketch an example with no common coding; assume the parameters are passed in GR0 and GR1:

```
Shifter  CSect ,                   Control section with 2 entry points
         ENTRY ShftRt,ShfLft       Declare the two entry points
         Using *,15                Caller presets R15
ShftRt   LTR   1,1                 Test right shift count
         BNM   ShfOKR              Branch if not negative
         SR    1,1                 Set to zero if it was negative
ShfOKR   SRL   0,2(1)              Shift right
         BR    14                  Return
         Using *,15                Caller presets R15
ShfLft   LTR   1,1                 Test left shift count
         BNM   ShfOKL              Branch if not negative
         SR    1,1                 Set to zero if it was negative
ShfOKL   SLL   0,2(1)              Shift left
         BR    14                  Return
         End   ,
```

Figure 570. Subroutine with entries for two similar functions

Now, we will rewrite the example to use some instructions common to the two subroutines. The routine isn't shorter, but illustrates a technique you can use when multiple entry points in a routine have common instruction sequences that can be shared by each entry.

```
Shifter  CSect ,                   Control section with 2 entry points
         ENTRY ShftRt,ShfLft       Declare the two entry points
         Using *,15                Caller sets base register
ShftRt   MVI   ShFlag,0            Set flag byte for right shift
         B     ShftAA              And enter common code
         Using *,15                Using for second entry point
ShfLft   MVI   ShFlag,1            Set flag byte for left shift
ShftAA   BASR  15,0                Reset base register for local use
         Using *,15                And set up correct using info
         LTR   1,1                 Test shift count
         BNM   ShftOK              Branch if not minus
         SR    1,1                 Otherwise set to zero
ShftOK   TM    ShFlag,1            Test direction of shift
         BZ    ShfR                Branch if 0, meaning right
         SLL   0,2(1)              Perform left shifts
         BR    14                  Return to caller
ShfR     SRL   0,2(1)              Perform right shifts
         BR    14                  Return to caller
ShFlag   DS    X                   Flag byte
         Drop  15
```

Figure 571. Subroutine with two similar functions and some common code

The BASR instruction named **ShftAA** and its associated USING are needed to establish a base register with a known value for the following instructions. If they had been omitted, the implied addresses of the subsequent instructions might use the wrong base. If the entry had been at **ShftRt** GR15 would contain its address, whereas the USING in effect later will assume that GR15 contains the address of **ShfLft**.

38.5.4. The External Symbol Dictionary Listing

When the assembly is complete, the ESD information is encoded into the object module and displayed in the Assembler listing's External Symbol Dictionary.[257] For example, if we assemble this little program:

[257] You can suppress this part of the listing by specifying the Assembler's NOESD option.

```
Main     Start X'2400'           Main control section
         DS    24D
         EXTRN SomeSym           External symbol declaration
ASection CSect ,                 Second control section
         DS    XL137
ASecEnt  Equ   *                 Declare internal symbol
         ENTRY ASecEnt           Declare internal entry point
         DS    XL131
RSection RSect ,                 Third control section
         DS    XL149
ComSect  COM   ,                 Fourth control section
         DS    200F
ADummy   DXD   CL44              Dummy external section
         End
```

Figure 572. Sample assembly with external symbols

the Assembler's External Symbol Dictionary listing looks like Figure 573:

```
                        External Symbol Dictionary

Symbol    Type   Id      Address  Length   Owner Id Flags

MAIN       SD 00000001 00002400 000000C0             00
SOMESYM    ER 00000002
ASECTION   SD 00000003 000024C0 0000010C             00
ASECENT    LD          00002549          00000003
RSECTION   SD 00000004 000025D0 00000095             08 ←— RSECT flag
COMSECT    CM 00000005 00000000 00000320             00
ADUMMY     XD 00000006 00000000 0000002C
```

Figure 573. External symbol dictionary from sample assembly

The seven fields in Figure 573 are:

1. Symbol: the external symbol, converted to upper-case letters.[258]
2. Type: the type of external symbol, described in Section 38.5 on page 818.
3. Id: the ESDID of the symbol. Note that the symbol **ASECENT** has no ESDID of its own, because it is in the control section **ASECTION**, identified by “Owner Id” X'00000003'.
4. Address: the location at which the control section begins. For entry names (identified by Type LD), the address is the location of the entry point within the “owning” control section.

   Note that each section address has been rounded up to the next doubleword boundary. For example, ASECTION starts at location X'24C0' and is X'10C' bytes long, so the first available location is X'25CC' and RSECTION starts at the next doubleword, X'25D0'.
5. Length: the length of the control section.
6. Owner Id: for entry names (Type LD), the ESDID of the control section in which the entry point resides.
7. Flags: bits indicating the addressing and residence modes and RSECT status declared for each control section. In this example, all have specified 24-bit addressing mode and residence below the 16MB “line”.

We can now give a complete definition of a symbol's relocation attribute:

[258] Upper-cased external symbols have roots in the days of System/360, when almost all programs were created on punched cards using upper-case letters. You can generate external symbols with mixed-case letters and other characters using the ALIAS instruction; see the *High Level Assembler Language Reference* for details.

1. If a symbol appears in the ESD only, or in the ESD *and* in the ordinary symbol table, its relocation attribute is its ESDID.
2. If a symbol appears only in the ordinary symbol table, its relocation attribute is the ESDID of the control section in which it appears. (Undefined symbols may have ESDID zero.)

A special Type indication (PC) appears in the Assembler's External Symbol Dictionary listing for unnamed control sections; PC stands for “Private Code”.[259] As noted on page 804, Private Code sections may be generated if you put some statements that could affect the Location Counter ahead of the first START or CSECT statement; the Assembler automatically creates a blank-named CSECT.

For example, this set of statements:

```
R1       Equ   1                 EQU can affect the Location Counter!
Main     Csect ,                 Start of a control section
         - - -                   Rest of 'Main' CSect
         End
```

could produce an External Symbol Dictionary listing like this:

```
Symbol   Type   Id     Address  Length

          PC 00000001 00000000 00000000   ←— Unnamed zero-length CSect
MAIN      SD 00000002 00000000 000002D4
```

Unfortunately, PC sections can sometimes cause the generated load module to have unintended addressing and residence modes.

ESD records describe four types of external symbols. These two-letter Type abbreviations identify the symbol's type:

SD,CM,PC Section Definition: the name of a control section, CM identifies COMmon sections having no machine language “text”, and PC identifies a blank-named control section; these three are doubleword aligned by default.

LD Label Definition entries identify the name of a position at a fixed offset *within* an “owning” Control Section. They are used to identify entry points. Because Label Definitions belong to another CSect, they are the only symbol type having no ESDID of their own.

ER,WX These two external symbol types are for External References to symbols not defined in this module, but to a symbol defined elsewhere to which this module wants to refer. A special form of external symbol is called WX or “Weak EXternal”; this type of reference might not be resolved at link time, without error.

XD The name of an “EXternal Dummy Section”; sometimes called a PseudoRegister (PR). XD names are in a separate link-time “name space” from all other external symbols, and may match non-XD external names without conflict. Section 38.7.3 will discuss their use.

If you specify the SECTALGN(16) option, three different Type identifiers are used for quadword-aligned control sections:

SQ for a quadword-aligned control section,

CQ for a quadword-aligned COMMON control section.

PQ for a quadword-aligned private (unnamed) control section,

To illustrate, we'll assemble the simple program in Figure 574 on page 827 with each SECTALGN option:

[259] “Private” because you can't refer to such a control section by its nonexistent section name!

```
         CSect ,                Blank CSect name (Private Code)
         DS    XL5
Section  CSect ,                Named control section
         DS    XL7
Common   COM   ,                Common section
         DS    XL3
         End
```

Figure 574. Program assembled with different SECTALGN options

Then, Figure 575 shows the key parts of the External Symbol Dictionary listing with each option; only the Type and Address columns are different.

```
          With SECTALGN(8)           |          With SECTALGN(16)
                                     |
 Symbol   Type Id   Address Length   | Symbol   Type Id   Address Length
            PC 00001 0000000 0000005 |            PQ 00001 0000000 0000005
 SECTION    SD 00002 0000008 0000007 | SECTION    SQ 00002 0000010 0000007
 COMMON     CM 00003 0000000 0000003 | COMMON     CQ 00003 0000000 0000003
```

Figure 575. Example of ESD listings with different SECTALGN options

The forms of ESD data in an object module, and how each is processed, will be discussed starting in Section 38.6.

38.5.5. External Symbol Addressing and Residence Modes

External symbols defining executable control sections (START, CSECT, and RSECT) or COM reference sections can be assigned *mode* attributes specifying where in memory you want the section to be loaded (its *residence mode* or RMODE), and what addressing mode (its AMODE) should be assigned by the Program Loader when it passes control to that symbol (assuming the symbol is designated as the program's entry point). These attributes are assigned by the AMODE and RMODE assembler instruction statements.

The allowed values of AMODE and their meanings are shown in Table 404.

AMODE	Meaning
24	The instructions in this control section, or at this entry point, should receive control in 24-bit addressing mode.
31	The instructions in this control section, or at this entry point, should receive control in 31-bit addressing mode.
64	The instructions in this control section, or at this entry point, should receive control in 64-bit addressing mode.
ANY, ANY31	The instructions in this control section, or at this entry point, may receive control in 24-bit or 31-bit addressing mode.
ANY64	The instructions in this control section, or at this entry point, may receive control in any addressing mode.

Table 404. AMODE values

The allowed values of RMODE and their meanings are shown in Table 405 on page 828. (For an illustration of the 16MB “line” and the 2GB “bar”, see Figure 149 on page 308.)

RMODE	Meaning
24	The control section should be loaded below the 16MB "line".
31, ANY	The control section should be loaded below the 2GB "bar", either above or below 16MB
64	The control section may be loaded anywhere in memory.

Table 405. RMODE values

If either AMODE or RMODE (or neither) is specified for a section, the Assembler assigns the values shown in Table 406.

Declared Mode	Assigned Mode
none	AMODE 24, RMODE 24
AMODE 24, AMODE 31, AMODE ANY, AMODE ANY31	RMODE 24
RMODE 24	AMODE 24
RMODE 31, RMODE ANY	AMODE 31
AMODE 64, AMODE ANY64	RMODE 31
RMODE 64	AMODE 64

Table 406. Default AMODE and RMODE values

Only certain combinations of AMODE and RMODE values are valid, as shown in Table 407.

Mode	Valid values
AMODE 24	RMODE 24
AMODE 31, AMODE ANY, AMODE ANY31	RMODE 24, RMODE 31
AMODE 64, AMODE ANY64	RMODE 24, RMODE 31, RMODE 64

Table 407. Valid combinations of AMODE and RMODE values

Figure 576 shows how mode values are assigned to external symbols:

```
SR24A24  CSect ,
SR24A24  RMODE 24               Assign residence mode 24
SR24A24  AMODE 24               Assign addressing mode 24
         STM   14,12,12(13)     Entry point: save registers, ...
         - - -
SR24A31  CSect ,
SR24A31  RMODE 24               Assign residence mode 24
SR24A31  AMODE 31               Assign addressing mode 31
         DR    2,11
         - - -
SR31A31  CSect ,
SR31A31  RMODE 31               Assign residence mode 31
SR31A31  AMODE 31               Assign addressing mode 31
         - - -
SR31A64  CSect ,
SR31A64  RMODE 31               Assign residence mode 31
SR31A64  AMODE 64               Assign addressing mode 64
         - - -
```

Figure 576. Assigning RMODE and AMODE to a section name

The ESD listing from assembling this little fragment is shown in Figure 577 on page 829:

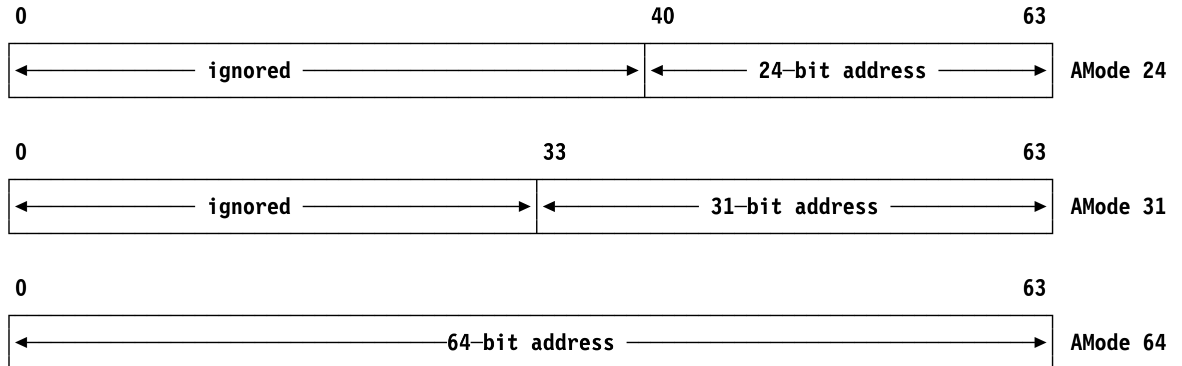
```
Symbol   Type   Id     Address  Length   Owner Id Flags

SR24A24   SD 00000001 00000000 00000004            01
SR24A31   SD 00000002 00000008 00000002            02
SR31A31   SD 00000003 00000010 00000000            06
SR31A64   SD 00000004 00000010 00000000            14
```

Figure 577. ESD showing RMODE and AMODE of section names

To help you understand the "Flags" field, do Exercise 38.5.17.

The addressing mode in effect when your program runs determines which register bits are used to form Effective Addresses, as shown in the following figures:



Exercises

38.5.1.(2)+ Assemble the program in Figure 572 on page 825 with the NOTHREAD option. Which values in the ESD portion of the listing have different values from those in Figure 573 on page 825? Why?

38.5.2.(1)+ Which assembler instruction statements define define executable control sections and which define reference sections? Which define space in the object module?

38.5.3.(2)+ Rewrite Figure 571 on page 824 so that the MVI instructions modify the *operation code* of a single shift instruction, so that it will perform the shifts in the desired direction. Remove all references to the flag byte.

Note! This is a *very* poor coding technique; this exercise is included only to show (what were believed to be clever) techniques used in "olden days" that now cause programs to run much more slowly.

38.5.4.(1) Why would specifying the SECTALGN(16) option be useful?

38.5.5.(1) Can you think of any use for Private Code sections?

38.5.6.(2) Write, assemble, and link a small program with unnamed CSECT and COM statements. Describe what happens.

38.5.7.(4) In Exercise 38.5.6 you wrote and assembled a small program with unnamed CSECT and COM statements. How could you refer to each from separately assembled modules?

38.5.8.(2)+ Determine whether the following variation on Figure 571 on page 824 will work correctly, and correct it if you think it won't. Explain your conclusions in either case.

```
         Using *,15
ShftRt   MVI   ShFlag,0          Set flag to zero
         B     SHFTAA            Branch to common code
ShfLft   LA    15,ShftRt         Set common base register
         MVI   ShFlag,1          Set flag for left shift
SHFTAA   LTR   1,1               ...etc...
```

38.5.9.(4)+ Suppose a programmer wrote Figure 571 on page 824 without the BASR at **ShftAA** and the following USING, and then attached the name **ShftAA** to the LTR instruction. Compute bases and displacements for all the instructions in the program (or, assemble it if necessary), and determine exactly what happens for correct calls to **ShftRt** and **ShfLft**.

38.5.10.(2)+ Write an external subroutine named **BYTE** with two arguments passed using standard conventions. The first argument is a byte, and the second argument is an 8-byte area where the subroutine will place 8 EBCDIC zero and one characters representing the bits of the byte argument.

38.5.11.(2)+ Write a separately-assembled subroutine named **SIGNUM** with a word integer argument that returns in GR0 the value −1 if the argument is negative, +1 if the argument is strictly positive, and 0 if the argument is zero. Assume standard linkage conventions, and restore all modified registers except GR0.

38.5.12.(2)+ In Figure 568 on page 822, the first Load instruction places the address of **Data** in GR6. Why can't it be replaced by this instruction?

```
         LA    6,Data
```

38.5.13.(3) Suppose you are writing a large program that needs an internal debugging aid that will dump areas of memory in hexadecimal in some readable format. Write a separately-assembled subroutine named **MemDump** with this standard calling sequence:

```
         L     15,=V(MemDump)
         CNOP  2,4               Align to halfword boundary
         BASR  14,15
         DC    A(start_address)
         DC    A(end_address)
```

You can use the PRINTLIN macro instruction to print the formatted lines.

38.5.14.(3) Revise your solution to Exercise 38.5.13 to allow either one or two memory addresses to be specified in the list following the BASR instruction, in this form:

```
         BASR  14,15
         DC    A(X'80000000'+start_address) For one argument
or
         BASR  14,15
         DC    A(start_address)
         DC    A(X'80000000'+end_address)   For two arguments
```

where the high-order bit indicates the last memory address. If only a single argument is given, display 32 bytes starting at the `start_address`.

38.5.15.(3) Write an external subroutine named **I2D** that converts the signed 32-bit integer value in GR0 to a long hexadecimal floating-point value in FPR0.

38.5.16.(3) Write an external subroutine that converts the short hexadecimal floating-point number in FPR0 to a fullword integer in GR0. If the result is valid, set return code 0 in GR15. If the floating-point number is too large, return the maximum negative number in GR0 and set return code 4 in GR15.

38.5.17.(3)+ The rightmost six bits of the “Flags” field indicate the AMODE and RMODE associated with the external symbol (remember that bit numbering starts at zero):

```
Bit 2 (R bit):        1 = RMODE 64; 0 = use "r" bit 5
Bit 3 (A bit):        1 = AMODE 64; 0 = use "aa" bits 6 and 7
Bit 4 (s bit):        1 = RSECT
Bit 5 (r bit):        1 = RMODE 31
Bits 6,7 (aa bits):  00 = AMODE 24 (default)
Bits 6,7:            01 = AMODE 24 (declared)
Bits 6,7:            10 = AMODE 31
Bits 6,7:            11 = AMODE ANY
```

Using the information in Table 407 on page 828, determine all valid values of the "Flags" field.

38.5.18.(1)+ Write an instruction sequence to call the **SubShf2** subroutine in Figure 566 on page 822.

38.6. Object Modules

While you don't usually need to know what's in your object modules, every executable program begins as an (un-executable) object module that must be transformed to a loadable/executable format. Any functional limitations in what can go into an object module[260] will also limit how we can think about and build our programs.

We'll first look at how the Assembler generates object modules from your source program, and then in Section 38.7 at how object modules are linked into executable load modules. Finally, in Section 38.8 we'll see how load modules and program objects are loaded into storage for execution.

The traditional ("OBJ") object module consists of 80-byte card-image records, with `X'02'` in column 1, and an identifying 3-character "tag" in columns 2-4. The 3-character tags are:

SYM This identifies records describing the internal symbols of your program. SYM records are rarely used now.

ESD ESD records contain the external symbols of your program and their types. Each symbol (except LD) is identified by an ID number called its "External Symbol Dictionary ID", or ESDID. Each ESD entry for a control section specifies its length and starting address.

The four types of external symbols are:

- Section Definition: the name of a control section
- Label Definition: the name of an entry point
- External Reference and Weak EXternal reference
- EXternal Dummy section

TXT These records contain the machine language instructions and data (the "Text") of your program. Each record indicates how many bytes of data it contains (the length of the data), which control section it belongs to (its "position ID", the ESDID of the control section owning the text), and where within that control section it goes (its starting address).

There are no "gaps" in the data on TXT records, even though there may have been uninitialized gaps visible in the listing.

RLD Relocation Dictionary records contain data about each relocatable address constant in your program: its type, where it is within its control section (its "Position ID", identified by the control section's ESDID), its location within that control section (its address), and what should be put in that field (specified by the ESDID of another external symbol, its "Relocation ID").

RLD records encode information about four types of address constant:

[260] The format of an object module is described in detail in the *High Level Assembler Programmer's Guide.* The GOFF ("Generalized Object File Format") object module allows more freedom than the traditional object module format; it is described in Section 38.8.1., and documented in the *MVS Program Management Advanced Facilities* manual.

1. A-type: resolves directly to the target address.
2. V-type: resolves directly to the target address, *or* to an indirect linkage assist.
3. Q-type: resolves to an offset value provided by the Linker (used for referring to external dummy sections).
4. Cumulative External Dummy: a length constant; the Linker inserts the total length of all external dummy sections in the bound program. (More about this in Section 38.7.3.)

END The END record is the last record of an object module, and contains some additional "IDR" data identifying the program that created the object module. If your source program specifies an operand on the END statement (as in Figure 33 on page 81) requesting that the executed program receive control at that operand, that information is also encoded on this record.

There is at least one control section per object module, and one object module per assembly. If you specify the Assembler's "BATCH" option, one invocation of the Assembler may produce multiple object modules.

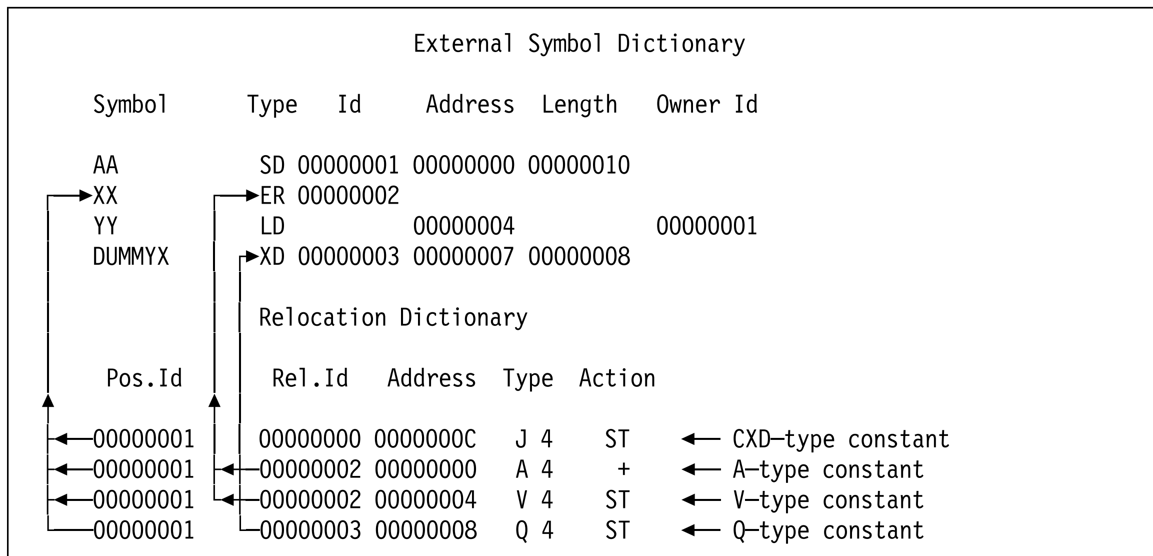
38.6.1. Relocation Dictionary and External Symbol Dictionary

Suppose we assemble the statements in the following figure; its listing extract shows how the Assembler describes relocation information:

```
  Loc  Object Code  ...  Stmt   Source Statement

000000              ...     1 AA      CSect ,          Start of a control section
                    ...     2         EXTRN XX         External symbol
                    ...     3         ENTRY YY         Entry name
                    ...     4 DummyX  DXD   D          Dummy External Section
000000 00000000     ...     5         DC    A(XX)      A-type constant
000004 00000000     ...     6 YY      DC    V(XX)      V-type constant
000008 00000000     ...     7         DC    Q(DummyX)  Q-type constant
00000C 00000000     ...     8         CXD   ,          Cumulative external dummy size
                    ...     9         End   ,
```

The Assembler's External Symbol and Relocation Dictionaries in the listing look like this:



```
                              External Symbol Dictionary

      Symbol      Type   Id      Address  Length   Owner Id

      AA           SD 00000001 00000000 00000010
      XX           ER 00000002
      YY           LD          00000004            00000001
      DUMMYX       XD 00000003 00000007 00000008

                   Relocation Dictionary

       Pos.Id       Rel.Id   Address  Type  Action

      00000001     00000000 0000000C   J 4    ST    <-- CXD-type constant
      00000001     00000002 00000000   A 4     +    <-- A-type constant
      00000001     00000002 00000004   V 4    ST    <-- V-type constant
      00000001     00000003 00000008   Q 4    ST    <-- Q-type constant
```

- All the RLD items have Position ID ("Pos. Id") `X'00000001'`, meaning that all the adcons are found in control section **AA** with that ESDID.
- The Relocation ID ("Rel. Id") gives the ESDID with respect to which the relocation will be done.

For example, both the A-type and V-type adcons are relocated with respect to the symbol **XX** which has ESDID 2. The Q-type constant relocates with respect to the symbol **DUMMYX** which has ESDID 3. CXD items have Relocation ID zero.

- The Address field shows the position within the owning control section (identified by the Position ID) where the adcon starts.

  For example, the V-type adcon resides in section **AA** with ESDID 1, at location X'00000004'.

- The Type field indicates the type and length of the adcon. Type J is used for length constants.
- The Action field indicates whether the relocation value will be added (+) or subtracted (-) from the contents of the field in the object text identified by the Position ID and Address, or whether the relocation value is *stored* (ST) in the object-text field.

Exercises

38.6.1.(4)+ Assemble this little program with the OBJECT option.

```
Prog     CSect ,                   Control section name
         Extrn X                   External name
         Entry B                   Entry name
         DC    F'-987'             Constant
B        DC    A(X,PRVLen)         A-type address constant
         DC    V(Y)                V-type address constant
D        DXD   3H                  External dummy section item
         DC    Q(D)                Offset of D
PRVLen   CXD   ,                   Cumulative external dummy length
C        COM   ,                   Common section
         DS    4F                  Space in the common section
         End
```

Use any convenient method to display the object module in hexadecimal format. With the High Level Assembler Programmer's Guide as a guide, study the object module to understand how the ESD, TXT, and RLD records are encoded.

38.6.2.(2)+ What are the differences in the effects of these two types of statements?

```
         DC    V(X)         and            EXTRN X
                                           DC    A(X)
```

38.7. Program Linking: Combining Object Modules

We will use a simple example to show how two object modules are linked and loaded directly into memory. The principles are very similar to creating a load module; we'll explain the differences in Section 38.8. Suppose a program consists of these two source modules:

```
     Module 1                                 Module 2
Loc ┌─────────────────────────┐          Loc ┌─────────────────────────┐
000 │MAIN    Start 0          │          000 │SUB     Start 0          │
    │        - - -            │              │        - - -            │
    │        CALL  SUB        │              │        EXTRN XDATA      │
    │        - - -            │              │        - - -            │
200 │ASub    DC    A(Sub)     │          700 │AWork   DC    A(Work)    │
204 │ACom    DC    A(Work)    │          704 │AXData  DC    A(XDATA)   │
208 │ADat    DC    A(XData)   │              │        - - -            │
    │        - - -            │              │Work    COM   ,          │
    │        Entry XData      │              │        DS    XL(X'400') │
    │        - - -            │              │        End              │
260 │XData   DC    160X'01'   │              └─────────────────────────┘
    │        - - -            │
    │Work    COM   ,          │
    │        DS    XL(X'600') │
    │        End   MAIN       │
    └─────────────────────────┘
```

Figure 578. Example of two source modules to be linked

- Program **MAIN** contains the **XDATA** entry point, and refers to the subroutine **SUB** and the common control section **WORK**, which it requires to be X'600' bytes long.
- Subroutine **SUB** refers to the external name **XDATA** and to the common control section **WORK**, which it requires to be X'400' bytes long.

Assembling the source modules produces two object modules. The object module for Module 1 would look roughly like this:

```
┌──────────────────────────────────────────────┐
│ESD SD ID=1 MAIN  Addr=000 Len=300            │  SD for CSECT MAIN, ESDID=1, Len=300
│ESD CM ID=2 WORK  Addr=000 Len=600            │  CM for COMMON WORK, ESDID=2, Len=600
│ESD LD ID=1 XDATA Addr=260                    │  LD for Entry XDATA, ESDID=1, Addr=260
│ESD ER ID=3 SUB                               │  ER for reference to SUB, ESDID=3
│TXT    ID=1 Addr=000      'abcdefghijk...'    │  Text in MAIN, address 000
│TXT    ID=1 ...            etc.               │  Text in MAIN
│TXT    ID=1 Addr=100      'mnopqrstuvw...'    │  Text in MAIN, address 100
│TXT    ID=1 Addr=208       00000260           │  Text in MAIN, internal adcon offset
│TXT    ID=1 Addr=260      '010101010101...'   │  Text in MAIN, address 260
│TXT    ID=1 ...            etc.               │  Text in MAIN
│RLD    PID=1 RID=3 Addr=200 Len=4 Type=V Dir=+│  RLD item for Addr(SUB)
│RLD    PID=1 RID=2 Addr=204 Len=4 Type=A Dir=+│  RLD item for Addr(WORK)
│RLD    PID=1 RID=1 Addr=208 Len=4 Type=A Dir=+│  RLD item for Addr(XDATA)
│END    Entry=MAIN                             │  Module end; request entry at MAIN
└──────────────────────────────────────────────┘
```

Figure 579. Sketch of object module from source module 1

The object module contains:

- ESD records for two control sections (**MAIN** and **WORK**), one entry (**XDATA**), and one external reference (to **SUB**).
- RLD records with information about the three address constants. “PID” represents the Position ID, “RID” the Relocation ID, and “Dir” indicates whether the relocation value will be added or subtracted.
- TXT records with machine language instructions and data. The TXT record for Addr(XDATA) contains the location (X'00000260') within control section **MAIN** because the target of the adcon is internal to the section.

The object module for Module 2 would look roughly like this:

```
ESD SD ID=1 SUB   Addr=000 Len=800                    SD for CSECT SUB, ESDID=1, Len=800
ESD CM ID=2 WORK  Addr=000 Len=400                    CM for COMMON WORK, ESDID=2, Len=400
ESD ER ID=3 XDATA                                     ER for reference to XDATA, ESDID=3
TXT    ID=1 Addr=040      'qweruiopasd...'            Text in SUB, address 040
TXT    ID=1 ...            etc.                       Text in SUB
TXT    ID=1 Addr=180      'jklzxcvbnm...'             Text in SUB, address 180
TXT    ID=1 ...            etc.                       Text in SUB
RLD    PID=1 RID=2 Addr=700 Len=4 Type=A Dir=+        RLD item for Addr(WORK)
RLD    PID=1 RID=3 Addr=704 Len=4 Type=A Dir=+        RLD item for Addr(XDATA)
END                                                   Module end; IDR data
```

Figure 580. Sketch of object module from source module 2

- The ESD records define two control sections (**SUB** and **WORK**) and one external reference (to **XDATA**).
- The RLD records contain information about two address constants.

Note that both object modules assign ESDIDs starting at 1.

First, the Linker/Loader must reserve some memory to hold the loaded program; we'll suppose that block of memory starts at X'00123500'; this is called the *load address*.

As the object modules are read by the Linker/loader, the external symbols are entered into a "Composite External Symbol Dictionary" (CESD). It will contain all external symbols used in each of the object modules being linked and loaded. After the first object module has been read and the TXT data has been moved into memory starting at the load address, the CESD would look like this:

Symbol	Type	ESDID	Address	Length
MAIN	SD	0001	123500	300
WORK	CM	0002		600
XDATA	LD	*0001	123760	
SUB	ER	0003		

Figure 581. Composite ESD after reading first object module

In the CESD entry for **XDATA**, the * on the ESDID means that 0001 is its "owning" section's ESDID. That is, **XDATA** is an entry point in control section **MAIN**.

When the second object module is read, its ESDIDs *also* start at 1. For those of its ESDIDs that differ from symbols already in the CESD, new ESDIDs must be assigned to prevent conflicts; this process is called "renumbering". The Position and Relocation IDs of address constants are also renumbered.

The Linker adds new external names to the CESD (in this case, there are no new names) and adjusts the start address of **SUB** to start at the next doubleword boundary after the end of **MAIN**. The CESD would then look like this:

Symbol	Type	ESDID	Address	Length
MAIN	SD	0001	123500	300
WORK	CM	0002		600
XDATA	LD	*0001	123760	
SUB	SD	0003	123800	800
SUB	ER	0003	123800	

Figure 582. Composite ESD after loading second object module

The symbol **SUB** appears twice in the Composite ESD because it is both a section definition and an external reference.

38.7.1. Assigning COMMON Sections

When reading the ESD of the second object module, the Linker notes that the requested length of the common block **WORK** is X'400', which is less than the X'600' length requested by the **MAIN** program. The Linker always retains the longest requested length for common sections, so the length X'600' remains unchanged.

The Linker can now assign an address to **WORK** at the next doubleword boundary following the end of **Sub**. The final, updated CESD then looks like this:

Symbol	Type	ESDID	Address	Length
MAIN	SD	0001	123500	300
WORK	CM	0002	124000	600
XDATA	LD	*0001	123760	
SUB	SD	0003	123800	800
SUB	ER	0003	123800	

Figure 583. Composite ESD after assigning memory addresses

38.7.2. Relocating Address Constants

We'll use the three adcons from sample program module 1 (see Figure 579 on page 834) to show how address constants are relocated.

1. A(XDATA):

   The Linker finds an RLD item referring to **XDATA**. Its position is in **MAIN** (ESDID=0001) at offset X'208', and it relocates relative to Relocation ID 0001 because the target is an entry in **MAIN**. Because it's an A-type constant, and the object text at the adcon's address contains X'00000260', the Linker loads that word. And because the Dir field indicates that the relocation value is to be added, the loader adds the relocation address X'123500' to the contents of the field at address X'123708' and stores the result, X'123760', back in **Main** at address X'123708'.

   The same process is used for the A(XDATA) adcon in routine **SUB**.

2. A(WORK):

The Linker finds an RLD item referring to **WORK**. Its position is in **MAIN** (ESDID=0001) at offset X'204', and it relocates relative to Relocation ID 0002. Because it's an A-type constant with no offset, the loader adds the relocation address X'124000' to the (zero) contents of the field at address X'123704' and stores the result back in **MAIN** at address X'123704'.

The same process is used for the adcon referencing **WORK** in subroutine **SUB**.

3. V(SUB):

   The Linker finds an RLD item referring to **SUB**. Its position is in **MAIN** (ESDID=0001) at offset X'200', and it relocates relative to Relocation ID 0003. Because a V-type constant cannot have an offset, the loader *stores* the relocation address X'128000' at address X'123700' in **MAIN**.

Now that addresses have been assigned to all external symbols, the address constants have been relocated by adding or subtracting the relocation value to or from each adcon's P-field contents (for A-type adcons), or by storing the relocation value in V-cons.

After loading and relocation are complete, the program in memory would look like this:

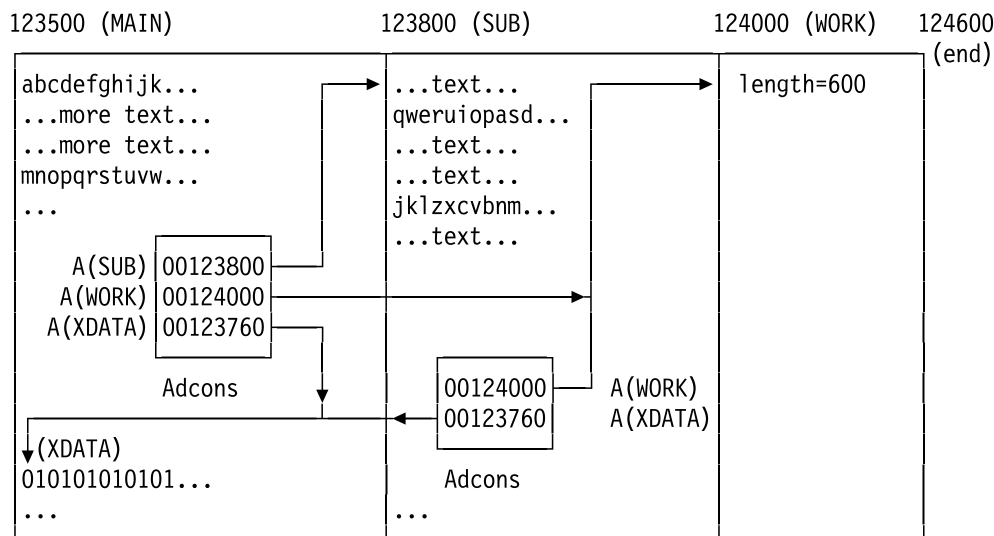


Figure 584. Memory layout of loaded program

To summarize:

- Storage was allocated for three control sections (two SD, one CM); any excess memory (including that for the CESD) is freed.
- Address constants were resolved to the designated addresses.
- The program was given control at the entry point requested by the END statement of the **MAIN** program, at address X'123500'.
- The loaded program is X'1100' bytes long.

This example has ignored what happens to unresolved strong external references after all object modules have been read. In practice, the Linker searches a “library” for names matching the remaining ERs and loads them in the same way it loaded the object modules. For example, if the **MAIN** program had contained an EXTRN for **COS** *and* **COS** was not defined in one of the input modules, the Linker will search the library for a member of that name. It's important that this search is done *after* all primary inputs have been processed, because you may have written your own **COS** routine!

38.7.3. External Dummy Sections (*)

External Dummy Sections[261] are used in situations like these:

- Separately assembled or compiled modules must share access to a resource by *name*, such as an output file like FILE1.
- The complete linked program may be loaded into memory once, but executed simultaneously by more than one invoker. Thus, it cannot safely modify any of its own internal areas. Such programs are called "reenterable".[262]

None of the programs declaring the name of an external dummy item component of an External Dummy Section defines the area of storage it names; the allocation of memory to the External Dummy Section is done at execution time, as we'll see.

Using an External Dummy Section requires actions at three different stages: during assembly, during linking, and at execution time. We'll see shortly why it's called an "External Dummy Section", and will give an example to show how these three stages work together.

Assembly time: You declare your external dummy item contributions to the External Dummy Section in one of two ways:

1. Write a DXD statement defining the name, length, and alignment of *your* external dummy item component of the External Dummy Section. This example defines two External Dummy Sections:

```
FILE1CB  DXD   A                 4 bytes, word aligned
RANDOM   DXD   D                 8 bytes, doubleword aligned
```

Figure 585. Sample DXD declarations

2. Define a Dummy control section (DSECT) and use its name in a Q-type address constant; this automatically makes the DSECT name an External Dummy Section:

```
TreeHead DSect ,                 Head of a binary tree for symbols
LftLink  DS    A
RgtLink  DS    A
SymLen   DS    X
Symbol   DS    CL63
```

Figure 586. External dummy section declaration

Then, define a Q-type constant referring to the DSECT or DXD name:

```
File1CBP DC    Q(FILE1CB)        External Dummy Section offset
RandomP  DC    Q(RANDOM)         External Dummy Section offset
TreeHdP  DC    Q(TreeHead)       External Dummy Section offset
```

Figure 587. Referencing external dummy items with Q-cons

The Assembler's External Symbol Dictionary entries for these three symbols are shown in this excerpt:

261 "Ordinary" Dummy Control Sections (DSects) will be discussed extensively in Chapter XI.

262 The more common term is "reentrant", but this also refers to a type of mathematical curve, which doesn't apply here.

```
Symbol    Type   Id       Address  Length

FILE1CB    XD 00000002 00000003 00000004
RANDOM     XD 00000003 00000007 00000008
TREEHEAD   XD 00000004 00000007 00000048
```

Figure 588. External dummy items in ESD listing

The “Address” field is actually an alignment mask: the requested alignment for the External Dummy Section item as a power of two, less one. (See Exercise 38.7.3.)

Link time: The Linker collects all the External Dummy Section contributions from the set of object modules it is linking, and constructs the template for the complete External Dummy Section. Each item is aligned according to its definition, and space is reserved according to its length. If more than one contribution has the same name, the Linker assigns the strictest alignment and longest length to that name. Thus, the Linker is creating something like a link-time Dummy Section; like a DSECT, it only describes an area of memory without actually allocating it.

For example, if a separate module also defined the DXD symbol **RANDOM** as in Figure 589,

```
RANDOM   DXD   L                  16 bytes, doubleword aligned
```

Figure 589. Separate DXD declaration

the Linker would use the longer of this value and the one in Figure 585 on page 838, and assign length 16 and doubleword alignment to that name.

The completed External Dummy Section might look like this:

```
   Offset

      0   |       - - -        |  other items...
          |--------------------|
     38   |FILE1CB   XD item   |  length 4, word aligned
     40   |RANDOM    XD item   |  length 16, doubleword aligned
          |--------------------|
     50   |       - - -        |
          |--------------------|
     6C   |TreeHead  XD item   |  length 72, word aligned
          |--------------------|
     B4   |       - - -        |
          |--------------------|
     DC                           Total length of the External Dummy Section
```

Figure 590. Example of a completed External Dummy Section

After the Linker has assigned all external dummy items offsets in the External Dummy Section, it then resolves the Q-type address constants by placing the *offset* from the start of the External Dummy Section of the operand name. Thus, for example, any Q-type address constant referring to **FILE1CB** would contain X'00000038'; if the Q-con had explicit length 2, it would contain X'0038'.

Finally, the Linker puts the accumulated length (X'000000DC') of the entire External Dummy Section into each CXD-type address constant. (You can now understand why CXD means “Cumulative External Dummy“. Usually, there's only one CXD in a completed program; we'll see why in a moment.)

When linking is complete, all address constants of types A, V, Q, and CXD have been resolved: A- and V-type for internal and external symbol references, Q-type for External Dummy Section offsets, and CXD-type for the cumulative length of the External Dummy Section.

Execution time: During initialization, each invocation of the main program uses operating system services to allocate an area of memory whose length is determined by the link-time value placed in the CXD-type constant, and sets the area to zeros. The address of this new area is put in a general register known to all routines that will access fields in the External Dummy Section.

Suppose GR11 has been assigned to hold this address:

```
         L    0,EDSLen        Get length of External Dummy Section
         GetMain R,LV=(0)     Get storage
         LR   11,1            Carry E.D.S. address in R11
         - - -                Initialize contents to zero
EDSLen   CXD  ,               Linker inserted total length of E.D.S.
```

Now, suppose one of several modules in the complete program wants to write records to FILE1, and has declared an external dummy item as in Figure 585 on page 838. It retrieves the FILE1CB item (a "file control block" giving access to the I/O routines) from the External Dummy Section:

```
         L    2,=Q(FILE1CB)  Get EDS offset of FILE1CB pointer
         AR   2,11           Storage address of FILE1CB pointer
         L    2,0(,2)        Pointer to FILE1CB now in R2
```

Figure 591. Retrieving an External Dummy Section item

The routine can now call the routine that writes records, passing the addresses of the record, its length, and the address of the FILE1CB field in the External Dummy Section (which will initially be zero). The write routine notes the zero FILE1CB field and allocates more storage to hold the control blocks and other items needed to complete the write operation. It then stores the address of these control blocks back in the FILE1CB field in the External Dummy Section.

Later, any other routine that wants to write records to FILE1 will retrieve the updated FILE1CB pointer from the External Dummy Section as in Figure 591; the write routine can then access the already-established control blocks to write the record.

Some history:

- External Dummy Sections and external dummy items were originally used in OS/360 for reenterable PL/I applications to allow sharing by name of dynamically managed external objects such as files, areas, and controlled variables that were defined in separately translated reenterable programs.
- PL/I called the external dummy items "PseudoRegisters" (PRs), and the External Dummy Section a "PseudoRegister Vector" (or "PRV"). Each PR item was a 4-byte address, so it was natural to think of them as representing additional "registers", and the External Dummy Section as a "vector" of PRs.
- Instructions referring to PR items were usually RX-type Load instructions. For example, the generated instruction was carefully arranged to do the same as in Figure 591, but saving one instruction:

```
         L     2,0(11,0)          Zero displacement and base fields!
         ORG   *-2
         DC    QL2(FILE1CB)       Displacement (and zero base)
```

Figure 592. PL/I technique for loading Pseudo Registers

  so that the generated instruction would be X'582B 0xxx' where 0xxx is the PR offset in the QL2-con, and the assigned External Dummy Section address (in GR11) is specified in the index register field of the L instruction. Thus, PL/I's PRV allowed up to 1024 more 32-bit "pseudo registers".

PseudoRegisters are not used frequently today.

COMMONs and DXD items have similarities and differences, as outlined in Table 408 on page 841.

	COMMON	External dummy item
Link-time behavior	Space allocated in the load module	No space allocated; a mapping of all items into a virtual External Dummy Section
Storage Allocation	Static: part of the load module	Dynamic: at execution time
Initialization	None	Execution-time responsibility
Copies	One per load module; not reenterable	One per reenterable load module, instantiated during each execution
External names	One per common, one per load module	One per item; no conflict with non-External Dummy Section names
Internal names	As many as you want	None (but you can map the item's inner structure with a DSECT)
References	Direct, with adcons	One level of indirection via Q-con offsets and the base register anchoring the allocated storage

Table 408. Differences in linking COMMONs and External dummy items

To summarize:

- An External Dummy Section is like a component of a DSECT, but no space allocated in the programs that define the external dummy items that compose it. It is a template, a data-structure mapping created at link time. (Hence the Assembler's name, “External Dummy”, as indicated by the XD type in the External Symbol Dictionary listing.)
- An external dummy item can have internal structure if it is mapped by a DSECT whose name appears in a Q-type constant.

38.7.4. Loading Object Modules (*)

To show how linkers load object modules, we will sketch the behavior of a simple loader; a real loader must handle complexities that we'll ignore here. The basic steps are

1. Clear the ESDID Translation Table at the start of reading each object module.
2. Get an external symbol.
3. Search the CESD to see if the symbol is already known from previous object modules.

For each input symbol type, search for a possible match among symbols of the types shown in Table 409. (During processing, LD types are sometimes “re-typed” as LR, meaning “Label Reference”.) As mentioned previously, PR symbols need not differ from the other types of symbols.

Input Type	Search These Types
SD	ER, LD, SD, CM
LD	ER, LD, SD, CM
ER	ER, LD, SD, CM
CM	ER, LD, SD, CM
PR	PR Only

Table 409. ESD symbol search types

Different processing is required, depending on whether the incoming symbol already exists in the CESD (“match processing”) or not (“no-match processing”).

No-Match Processing

If the incoming symbol is *not* found in the CESD, the loader takes these steps:

1. Make an entry in the ESDID Translation Table, unless the symbol is type LD (its ESDID is that of its owning section).

Indexed by ESDID ↓

Input ESDID	CESDID
1	5
2	17

← Newly assigned CESDID value for the new symbol identified by this ESDID (row 2)

Figure 593. ESDID Translation Table entry for an incoming symbol

2. Perform processing, depending on symbol type:

SD

a. Determine the next available memory location (by adding the lengths of all previous control sections to the initial load address, rounding up each CSect address (origin+Length) to the next doubleword boundary.

b. Enter the name and start address of the new CSect in its CESD entry. Assign a new translation ID in the ESDID translation table to convert the ESDID of the current entry to the newly assigned CESDID of the symbol, as shown in Figure 593. Chain RLDs in this section to the CESD entry.

A typical CESD entry might look like this:

Symbol	Type	Load address	Rel. Reloc. Const.	↑ RLDs
			Length, Alignment	← for PR items

Figure 594. A typical load-time CESD entry

where the Load Address is the true address assigned to the symbol, and the Relative Relocation Constant is used to adjust translator-assigned addresses to true addresses.

c. Test for table overflows, and terminate loading if any.

d. Determine the *relative relocation constant* from the expression

```
(Linker-assigned CSect address) - (translator-assigned address)
```

where the translator-assigned address in the address field of the ESD entry.

e. Set a flag if this is a zero-length CSect; otherwise add its length to the current load address and round up to the next doubleword boundary.

LD

a. Make a CESD entry for the symbol.

b. Determine the true address of the symbol by adding the relative relocation constant (address) of the CSect owning this LD item (the owning ID can be found in the CESD by using the ESDID Translation Table) to the input address of the LD item.

c. Make no ESDID Translation Table entry for this LD symbol (it has no separate ESDID entry).

ER Make a CESD entry for the symbol, and chain any associated RLD items to it.

CM

a. Make a CESD entry, and chain related RLDs to it. Set the Load Address to zero.
b. Enter the Common's length and input address to the CESD entry.

PR

a. Make a CESD entry for the symbol, enter the PR length and alignment, and chain related RLD items (Q-type constants) to it.

Match Processing

If the incoming symbol *is* found in the CESD, the loader must take more complex actions, depending on the types of the new and existing CESD symbols. Because Private Code (PC) names are not unique, treat each PC as a new, unique SD: make a Translation Table entry, and create a CESD entry with its load address, length, and relative relocation constant, and chain any associated RLD items to the entry.

(1) Existing CESD Symbol is SD

New Symbol	Processing
SD	**Error**: duplicate section. Set a flag to ignore TXT and RLD items for this ESDID.
CM	Check for CM length > SD length; error if so (CM references could overwrite subsequent CSects). Relocate RLDs referencing this symbol.
LD	**Error**: conflicting types.
ER	Make a Translation Table entry referring to the SD in the CESD.
PR	Ignore.

Table 410. Matching existing CESD SD symbol to incoming symbols

(2) Existing CESD Symbol is LD

New Symbol	Processing
SD	**Error**: ESDID SD symbol matches an existing entry point. Set a flag to ignore TXT and RLD items for this ESDID.
CM	**Error**: ESDID CM symbol matches an existing entry point.
LD	**Error**: matching symbols with conflicting types.
ER	Make a Translation Table entry referring to the CESD LD.
PR	Ignore.

Table 411. Matching existing CESD LD symbol to incoming symbols

(3) Existing CESD Symbol is CM

New Symbol	Processing
SD	Assign the greater of the lengths of the CESD CM and the incoming ESDID CM, and change the CESD CM symbol to SD.
CM	Assign the greater of the lengths of the CESD CM and the ESDID CM to the CESD CM.
LD	**Error**: matching symbols with conflicting types.
ER	Make a Translation Table entry referring to the CM.
PR	Ignore.

Table 412. Matching existing CESD CM symbol to incoming symbols

(4) Existing CESD Symbol is ER

New Symbol	Processing
SD	Change the CESD entry to SD, and update the cumulative length of the loaded program. Relocate RLDs chained to the ER item.
CM	Change the CESD entry to CM. Make a Translation Table entry for the ESDID CM.
LD	Change the CESD ER symbol to LD. Update the relocation constant to refer to the assigned LD address, and relocate RLDs.
ER	Make a Translation Table entry referring to the CESD ER symbol. If both ERs are WX, leave the WX flag on; otherwise set it off.
PR	Ignore.

Table 413. Matching existing CESD ER symbol to incoming symbols

(5) Existing CESD Symbol is PR

New Symbol	Processing
SD	Ignore.
CM	Ignore.
LD	Ignore.
ER	Ignore.
PR	Set the CESD length to the greater of the CESD and ESDID PR lengths, and the alignment to the stricter of the two alignments.

Table 414. Matching existing CESD ER symbol to incoming symbols

TXT Record Processing: The loader checks the ESDID of the TXT record; if invalid, it discards the record. If valid, it checks the length of the TXT data against the current load address to see if available storage would be exceeded, and terminates loading if so. Otherwise it moves the text to the assigned address, and updates the load address.

END Record Processing: If no entry-point address has been assigned by other means, and one is defined on this END record, save it. If an entry address has already been assigned, ignore the END record.[263]

Final Processing

1. **Unresolved ERs**: If any unresolved ER items remain, search the Load Library (if requested), and load and relocate the members whose names match the unresolved ER names.
2. **Commons**: After the text has been loaded, space for each CM section is allocated, checking the length against the amount of available storage. The storage address of each CM is updated in the CESD.
3. **PR Processing**: Knowing the length and alignment of each PR item, the loader assigns a virtual offset to each, respecting the requested alignments, and noting the maximum final offset. These offsets are used in relocating Q-type address constants referring to the PR symbols, and the total length (the maximum offset) is assigned to CXD-type address constants.
4. **RLD Processing**: The address of each symbol in the CESD is now known, so the RLD items assigned to each can be relocated. (Remember that the address of the operand of a V-type address constant is *stored* in its text field, while the address of the operand of an A-type address constant is *added* to its text field.)
5. Release working storage, and enter the loaded module at its entry address.

[263] If no entry-point address has been assigned at the end of the linking process, the Linker will usually assign the entry-point address to the first byte of the CSect at the lowest address.

Exercises

38.7.1.(1)+ In Figure 583 on page 836, why does the symbol **XDATA** have the same ESDID as the symbol **MAIN**?

38.7.2.(3) Common sections cannot have entry points. Suppose you are expected to write a routine that uses a symbol **YourData** that is defined at an unknown offset in another program's common section named **MyCom**, but you have no way of knowing in advance what that offset is, nor can you define an EXTRN for **YourData**.

Can you devise a way for the owner of the **MyCom** common section to help you determine the address of **YourData**? (It can be done!)

38.7.3.(2)+ Show how the Address fields in Figure 588 on page 839 can be used as alignment masks.

38.7.4.(2)+ The comment following Figure 592 on page 840 says that this technique allowed PL/I to define up to 1024 PRs. Why at most 1024, and not more?

38.8. Load Modules and Program Objects

We often want to save a linked program so it can be executed more than once without repeating the full linkage process. This is done by having the Linker create a *load module* or a *program object* (described in Section 38.2.2).

We will describe the load module format used on MVS-based operating systems (including z/OS). Other operating environments use different formats, but the general principles are similar.

An executable load module that can be loaded and executed many times is created following the same steps illustrated in Section 38.7, with these differences:

- Instead of allocating an area of memory and loading the object text directly into that area, the Linker builds a file with assumed origin *zero*. Thus, the first control section is assigned address X'000000', and subsequent control sections are added in order, following alignment of their offset from the origin.
- The completed load module file is written into a Partitioned Data Set (PDS) "library" from which it can be loaded for execution, or used as input to the Linker for relinking.
- The final assignment of memory addresses and adcon relocation happens when the Program Loader brings the load module into memory for execution; we'll describe that process in Section 38.9 on page 855.

Suppose we use the two object modules in Figure 579 on page 834 and Figure 580 on page 835 to create a load module. Now, all addresses in the CESD are assigned relative to a zero origin, as shown in Figure 595:

Symbol	Type	ESDID	Address	Length
MAIN	SD	0001	000000	300
WORK	CM	0002	000300	600
XDATA	LD	*0001	000260	
SUB	SD	0003	000900	800
SUB	ER	0003	000900	800

Figure 595. Composite ESD after assigning load module addresses

As in Figure 583 on page 836, the symbol **SUB** appears twice; in this case the ER belongs to the **MAIN** program, while the SD belongs to **SUB**. If the subroutine SUB is replaced by a newer version, you won't want the ER to be deleted because **MAIN** must still be able to call **SUB**.

The address fields of all RLD items are adjusted relative to zero. The text fields of adcons (such as those referencing **WORK** and **XDATA**) are updated to contain the offset relative to the zero assumed origin of their respective targets.

The length of the bound program is the same, X'1100' bytes.

As before, if there are unresolved external references the Linker will search one or more libraries and include them in the load module. When this process is completed, the load module is written to the library.

The records of a load module are similar to the records of an object module (which simplifies processing each):

SYM Object-module SYM records are copied directly into load modules. They are included only at user request.[264]

IDR Identification records (from object module END records, the Linker, the user, and other maintenance programs).

CESD The Composite External Symbol Dictionary; this is needed if the load module will be relinked.

TEXT Machine language instructions and data.

CTL/RLD Control records, for reading and relocating text records, and including Relocation Dictionary information.

EOM End of module (a flag field in a control record).

A sketch of a load module as it is written to the library is shown in Figure 596. (The "library" for load modules is a "Partitioned Data Set", or PDS.)

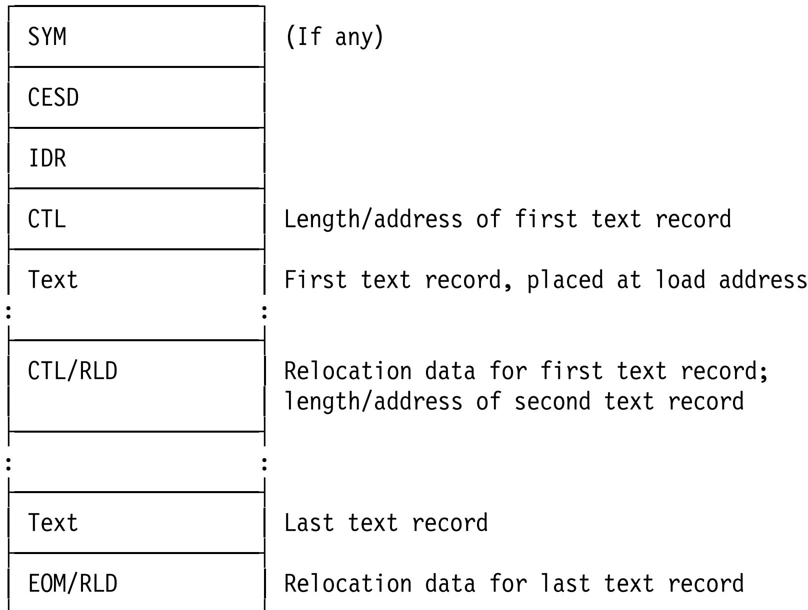


Figure 596. Sketch of a load module

[264] SYM records were originally used to help debuggers, but are rarely used now.

When loaded into memory, a load module is a *single* one-dimensional set of contiguous control sections: This means that a load module created from CSects with different RModes will necessarily be assigned *its* RMode from the lowest RMode of its components.

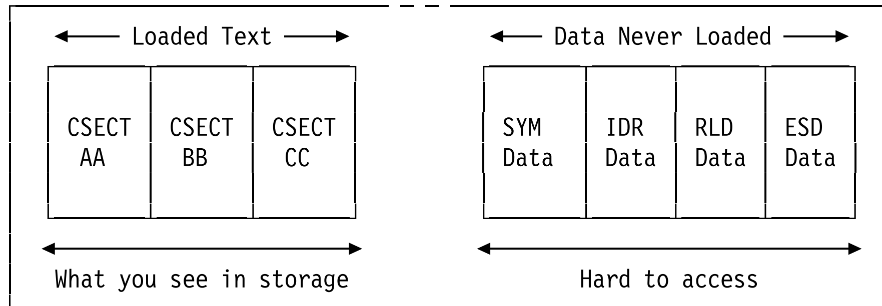


Figure 597. A load module after loading

The format of a load module was fixed very early in System/360 days, and many tools and products depended on its format not changing; this made it nearly impossible to add new function. The z/OS Binder now supports "Application Programming Interfaces" (APIs) that can extract and update any interesting information in a load module.

38.8.1. External Subroutines and Assisted Linkage: Overlay (*)

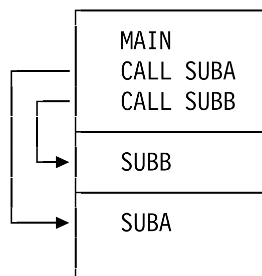
Assisted linkages with external subroutines are more complicated than for the internal subroutines we discussed in Section 37.6 on page 783, where uniform addressability was assumed.

If the routine providing the assisted linkage (such as the **Caller** routine in Figure 519 on page 784) is *external* to the routines making the calls, the subroutine numbers will have been defined there, so the values of subroutine numbers like **Print#** won't be known to the calling routines.

There are several ways to solve this problem, such as defining a table of subroutine numbers that can be copied into any program using **Caller**. Another is to create a macro instruction that defines the subroutine numbers, and any calling program can invoke the macro.

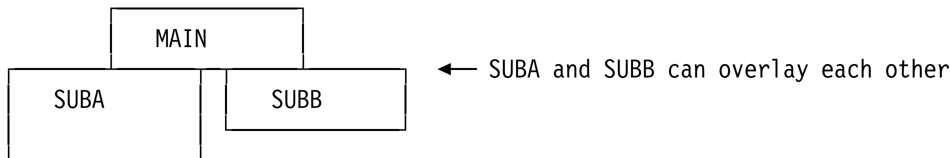
Overlay: An early technique for providing indirect linkage was called *overlay*. It was used extensively in programs that needed more memory than was economically available at the time.

For example, suppose you had a MAIN program that needed to call subroutines SUBA and SUBB, but neither subroutine was called by the other. A typical arrangement in storage might look like this, if there was enough room in memory:

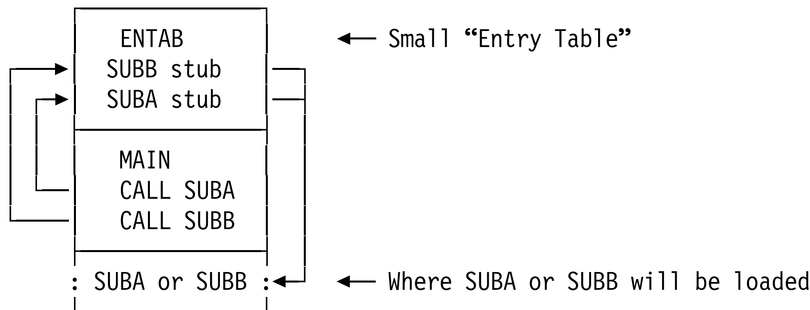


In this case, the V-type address constants in MAIN for each of SUBA and SUBB would be resolved directly to their targets.

But if there wasn't enough room for all three at the same time, you could arrange them like this, *if* you had a way to tell the Linker and Program Loader how to arrange the three routines:



The solution was provided by the Linker's "overlay" control statements. The Linker would arrange the routines so that the contents of memory looked like this when MAIN was loaded:



The Entry Table is constructed by the Linker, and contains short "stub" routines to which the V-type address constants in MAIN were resolved. Each stub called the Supervisor to load either SUBA or SUBB into memory if it wasn't already there. The stub also contained an A-type address constant with the memory address of the subroutine's entry point *where it would be after is was loaded*. The stub could then branch directly to the subroutine.

For example, suppose MAIN called SUBA first, so it was loaded into memory by the Supervisor. When SUBA returned control to MAIN it called SUBB; the Supervisor would then load SUBB into the same area of memory just occupied by SUBA, "overlaying" it.

If an area of working memory needed to be shared among MAIN, SUBA, and SUBB, the usual solution was to define a Common section that would be loaded with MAIN. Because its address would be known to all three routines, A-type address constants were used for direct references.

The cost of memory has dropped rapidly since the days when overlay was needed, so it's little used today. But it's an interesting example of how to manage assisted linkage among separately translated routines.

38.8.2. Program Objects (*)

Program objects are the newest form of z/OS loadable module.[265] A program object provides many enhancements compared to Load Modules: they can be much larger, portions can be loaded into distinct areas of memory, and they can retain many more forms of useful information that need not be loaded into memory along with machine instructions and data. The summary in Section 38.11 will compare these differences in greater detail.

A key feature of program objects is support for *classes*.[266] Whereas a load module is a collection of control sections loaded as a single entity into one area of memory, a program object is a collection of independently relocatable classes that can be loaded into multiple disjoint areas of memory. We'll describe program object loading in Section 38.9 on page 855.

You can create classes with the Assembler in several ways:

1. Generate traditional object modules as described in Section 38.6 on page 831. At link time, direct the Binder (the z/OS Linker) to save the generated result in a PDSE (Partitioned Data

[265] Some support is provided on z/CMS.

[266] These are *not* the same as classes as defined in object-oriented programming languages.

Set Extended), a more modern form of “library”. The Binder will create default classes for you. (Section 38.8.4 on page 854 explains how this is done.)

2. Specify the GOFF (Generalized Object File Format) option to the Assembler for the same source program. The Assembler will generate the new-format object file and assign default classes for you. The presence of classes will usually require the Binder to save the generated program object in a PDSE.
3. Specify the GOFF Assembler option, and create your own Classes using the CATTR (Class Attributes) Assembler instruction.

The first two of these ways to generate a program object require no changes to your source program.

Classes and Sections: It's easiest to think of the structure of a program object as having two dimensions, one defined by a Section and the other by a Class.

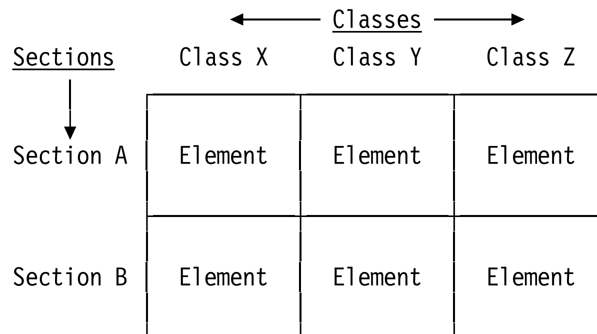


Figure 598. Sketch of program object structure

The term “Section” is used in a different sense with program objects. Rather than naming a control section (as with load modules), it names a set of contributions to one or more *Classes*. The contribution of a Section to a Class is called an “Element“. Elements are analogous to control sections from ordinary assemblies: they are an indivisible collection of machine language instructions and data.

Thus in Figure 598, Sections A and B each contribute instructions or data (or both) to Classes X, Y, and Z. A Section can contribute (or not) to any number of Classes.

Each Class in a program object has uniform binding and loading attributes and behavior; these are assigned when the Class is defined in your program.

As an example, suppose we assemble this little program and specify the GOFF option:

```
A       CSect ,
        STM   14,12,12(13)
*       - - -
B       CSect ,
        DC    3A(*-B+1)
        End
```

Figure 599. Sample program assembled with the GOFF option

Key parts of the External Symbol Dictionary listing from this assembly look like this:

```
Symbol   Type   Id     Address  Length   Owner Id
A        SD 00000001                            ←— Section
B_IDRL   ED 00000002                   00000001 ←— Class for IDR data
B_PRV    ED 00000003                   00000001 ←— Class for DXD items
B_TEXT   ED 00000004 00000000 00000004 00000001 ←— Class for object text
A        LD 00000005 00000000          00000004 ←— Entry
B        SD 00000006                            ←— Section
B_IDRL   ED 00000007                   00000006 ←— Class
B_PRV    ED 00000008                   00000006 ←— Class
B_TEXT   ED 00000009 00000008 0000000C 00000006 ←— Class
B        LD 0000000A 00000008          00000009 ←— Entry
```

Figure 600. ESD from program assembled with the GOFF option

There are many differences from what you'd see if the program had been assembled without the GOFF option:

- Although CSects A and B are shown as SD items, they have no address or length. They are simply "owners" or "containers" for the contributions to Classes.
- For each executable control section, the Assembler automatically generates three Classes, as indicated by the ED (for "Element Definition") in the Type column:[267]

  B_IDRL A Class to contain IDR (identification) data for each Section.

  B_PRV A Class to contain external dummy item data for each Section, in case any are generated.

  B_TEXT A Class that contains the generated object code for each Section. Each ED item in this class has an address (the starting address of the element) and a length. This is the default Class if no Class was explicitly defined by the CATTR instruction.
- Each ED item has its own ESDID, and its "Owner Id" is the ESDID of the section to which it belongs.
- Because you may want to refer to instruction or data items in a CSect by name, the Assembler generates an LD entry-point item for each control section's name at the origin of the B_TEXT Class into which the object code is generated. Thus, the LD entry for symbol A indicates it is at location 0 in the Class with ID 00000004, B_TEXT.
- Each LD item has an ESDID, because it can be assigned attributes not available without the GOFF option.
- Because program objects can be much larger than load modules (which are limited to 16MB in size), all address fields in the listing are 8 digits long rather than the 6 digits for an assembly without the GOFF option.

Similar comments apply to the External Symbol Dictionary entries generated by CSect B.

Because the little program in Figure 599 on page 849 (and others like it) don't specify more than a single Class for instructions and data, either a load module or a program object can be created by the z/OS Binder.

When you specify the GOFF option, you can assign AMODEs to entry (LD) and external (ER) symbols. For example:

[267] These three classes are generated so that older programs that define no explicit classes can be linked more easily by the Binder into program objects, giving compatible behavior between the load module and program object forms of the same program.

```
AModes   CSect ,
A        DC    V(B)             External symbol
A        AMODE 24
B        AMODE 64
         Extrn B
         Entry A                Entry point
```

Figure 601. Assigning AMODE to an entry symbol

The corresponding ESD listing is shown in Figure 602. (the AMODE for symbol **B** is properly encoded in the GOFF object file.)

```
Symbol   Type   Id     Address  Length   Owner Id Flags

AMODES    SD 00000001
B_IDRL    ED 00000002                    00000001
B_PRV     ED 00000003                    00000001
B_TEXT    ED 00000004 00000000 00000004  00000001   00
AMODES    LD 00000005 00000000           00000004   00
B         ER 00000006                    00000001
A         LD 00000007 00000000           00000004   01
```

Figure 602. ESD showing AMODE assigned to entry and external symbols

This technique can be useful if you create a module with more than one entry point, and wish to receive control in the intended addressing mode.

38.8.3. The "Class Attribute" Instruction CATTR

As sketched in Figure 598 on page 849, we need to specify both the Section *and* the Class to define the elements to which our data and instructions belong. The Section is defined by the traditional CSECT and RSECT instructions, and the Class is defined by the CATTR instruction. For example, the following assembly defines two Sections (SECT_A and SECT_B and three Classes (B_TEXT, CLASS_X, and CLASS_Y):

```
SECT_A   CSect ,
         STM   14,12,12(13)    Appears in Class B_TEXT
CLASS_X  CATTR ,
Msg_1    DC    C'A message'    Appears in Class CLASS_X
SECT_B   CSect ,
         LM    14,12,12(13)    Appears in Class B_TEXT
CLASS_X  CATTR ,
Msg_2    DC    C'Good News'    Appears in Class CLASS_X
SECT_A   CSect ,
CLASS_Y  CATTR ,
Msg_3    DC    C'Better News'  Appears in Class CLASS_Y
SECT_B   CSect ,
CLASS_Y  CATTR ,
Msg_4    DC    C'Enough!'      Appears in Class CLASS_Y
```

Figure 603. Sample program defining two Sections and three Classes

The assignment of instructions and data to the six elements defined by the Sections and Classes looks like this:

	B_TEXT	CLASS_X	CLASS_Y
SECT_A	STM	Msg_1	Msg_3
SECT_B	LM	Msg_2	Msg_4

Figure 604. Assignment of instructions and data into elements

The (slightly abbreviated) assembly listing of the program is shown in Figure 605 on page 852:

```
  Loc     ...  Stmt  Source Statement
00000000  ...     1 SECT_A   CSect ,
00000000  ...     2          STM   14,12,12(13)    Appears in Class B_TEXT
00000008  ...     3 CLASS_X  CATTR ,
00000008  ...     4 Msg_1    DC    C'A message'    Appears in Class CLASS_X
00000020  ...     5 SECT_B   CSect ,
00000020  ...     6          LM    14,12,12(13)    Appears in Class B_TEXT
00000011  ...     7 CLASS_X  CATTR ,
00000011  ...     8 Msg_2    DC    C'Good News'    Appears in Class CLASS_X
00000004  ...     9 SECT_A   CSect ,
00000028  ...    10 CLASS_Y  CATTR ,
00000028  ...    11 Msg_3    DC    C'Better News'  Appears in Class CLASS_Y
00000024  ...    12 SECT_B   CSect ,
00000033  ...    13 CLASS_Y  CATTR ,
00000033  ...    14 Msg_4    DC    C'Enough!'      Appears in Class CLASS_Y
```

Figure 605. Assembly listing for sample program

This listing excerpt shows several things:

1. The Location Counter values are 8 hexadecimal digits long. This is because portions of a program object can be larger than 16MB and can be loaded above the 16MB "line".
2. Each Class starts on a doubleword boundary. (You can use the ALIGN operand of the CATTR instruction to modify this.)
3. The contributions to CLASS_X are the constants named Msg_1 and Msg_2, both being 9 bytes long. Thus the contributions to CLASS_X total 18 bytes.
4. The contributions to CLASS_Y are the constants named Msg_3 and Msg_4, respectively 11 and 7 bytes long. Thus the contributions to CLASS_Y also total 18 bytes.

The External Symbol Dictionary created from assembling the program in Figure 599 on page 849 looks like this:

```
Symbol   Type   Id     Address  Length   Owner Id
SECT_A    SD 00000001                             ←— Section
B_IDRL    ED 00000002                    00000001 ←— Generated by Assembler
B_PRV     ED 00000003                    00000001 ←— Same
B_TEXT    ED 00000004 00000000 00000004  00000001 ←— STM instruction
SECT_A    LD 00000005 00000000           00000004 ←— SECT_A entry
CLASS_X   ED 00000006 00000008 00000012  00000001 ←— Class
SECT_B    SD 00000007                             ←— Section
B_IDRL    ED 00000008                    00000007
B_PRV     ED 00000009                    00000007
B_TEXT    ED 0000000A 00000020 00000004  00000007 ←— LM instruction
SECT_B    LD 0000000B 00000020           0000000A ←— SECT_B entry
CLASS_Y   ED 0000000C 00000028 00000012  00000007
```

Figure 606. External symbol dictionary for sample program

From this ESD listing, we see that:

1. The address assigned to each Class starts on a doubleword boundary.
2. The lengths of classes CLASS_X and CLASS_Y are both X'00000012' or 18 bytes.
3. The contributions to Class B_TEXT are each 4 bytes long.

The CATTR instruction supports several operands; the most useful are the RMODE and loadability attributes:

RMODE(n) This allows you to specify the region of memory where the Program Loader should place the Class. The allowed values of n are 24, 31, and 64.

ALIGN(n) You can specify other than the default alignment for the Class. The value of n is an exponent of 2 and takes values from 0 to 12, meaning byte to page alignment.

DEFLOAD A Class with this attribute can be loaded by the Program Loader when requested. It is used by reenterable programs to contain data and instructions needed for each of many simultaneous executions of the same program.

NOLOAD A Class with this attribute will be included in the program object by the Binder, but will not be loaded by the Program Loader. NOLOAD classes are typically used for symbol tables, source code, and other record-oriented information you want to keep with the executable program.

If neither DEFLOAD nor NOLOAD operands are specified, the default is LOAD, meaning that the class will be loaded when the program object is put in memory by the Program Loader.

PART(name) A Part is a named subdivision of an element that can have internal structure and be addressed by its name.

An example of defining a PART is shown in Figure 607:

```
PartSect CSect ,
ClassA   CATTR Part(P1)
         DC    2D'1.23'
ClassA   CATTR Part(P2)
         DC    C'Hello!'
```

Figure 607. Example of declaring parts in a GOFF Class

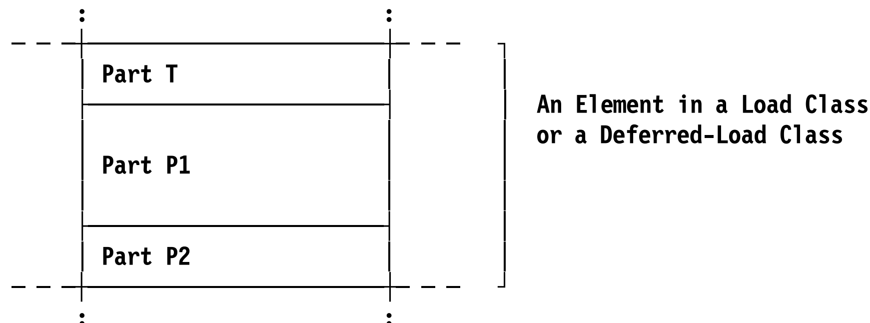
The corresponding ESD is shown in Figure 608; note that Parts are given type PD by the Assembler.

```
Symbol   Type   Id     Address  Length   Owner Id Flags

PARTSECT  SD 00000001
B_IDRL    ED 00000002                    00000001
B_PRV     ED 00000003                    00000001
B_TEXT    ED 00000004 00000000 00000000  00000001   00
PARTSECT  LD 00000005 00000000           00000004   00
CLASSA    ED 00000006 00000000 00000000  00000001
P1        PD 00000007 00000000 00000010  00000006   00
P2        PD 00000008 00000000 00000006  00000006   00
```

Figure 608. ESD for parts in a GOFF Class

An Element in a given Section and Class may contain several Parts; you can visualize them like this:



38.8.4. Programming for Program Objects

When writing programs that utilize classes and parts:

1. You must specify the GOFF option. If not, CATTR statements are simply checked for correct syntax, and are otherwise ignored.
2. Literals are generated wherever a LTORG statement appears; be careful where you write your LTORGs.
3. If any literals appear in your program after the last LTORG statement, check carefully that the Assembler generates them where you intended.
4. To resume a class, use its name in a CATTR statement with no operands. LOCTR statements can't be used to resume a class.
5. Be careful when using ORG with a Class or Part name, especially if the Class name appears in more than one section.
6. You can specify an AMODE for entry point names and external references that can be used in some situations to help set the correct addressing mode for references to such symbols. For example:

```
Prog     CSect ,
         DC    C'1'
ClassA   CATTR ,
XA       DC    C'2'
         ENTRY XA
XA       AMODE 31       ◄── Entry XA has AMODE 31
         EXTRN XB
XB       AMODE 64       ◄── External symbol XB has AMODE 64
```

The ESD listing from assembling this little program with the GOFF option is:

```
Symbol   Type   Id     Address  Length   Owner Id Flags

PROG      SD 00000001
B_IDRL    ED 00000002                    00000001
B_PRV     ED 00000003                    00000001
B_TEXT    ED 00000004 00000000 00000001  00000001   00
PROG      LD 00000005 00000000           00000004   00
CLASSA    ED 00000006 00000008 00000001  00000001
XA        LD 00000007 00000008           00000006   02 ◄── AMODE 31
XB        ER 00000008                    00000001
```

Even though the AMODE of symbol **XB** isn't shown in the "Flags" column, it is properly defined in the GOFF object file.

38.8.5. Comparing Load Modules and Program Objects

The z/OS Binder can create either a load module or a program object from old OBJ-format object modules. If the generated executable is a program object, the Binder makes a few internal changes:

- SD items: in addition to the control section name, the Binder creates an ED (Element Definition) for Class B_TEXT with the RMODE of the section, and an LD (Label Definition) at the origin of this element with the AMODE attribute of the original control section.
- CM (Common) sections are treated almost like SD items, and a "Common" flag is set.
- TXT records for the control section are assigned to the B_TEXT element. (There are no TXT items for Common sections.)
- XD items cause the Binder to create an ED item for the B_PRV class.

In creating a program object, the Binder may note that several classes may have identical binding and loading attributes, and combine the elements in those classes into single segments, to reduce loading time. The segments inherit boundary alignments of member classes.

From GOFF or OBJ object modules, the Binder can create either a load module or a program object, but a generated load module is necessarily limited to a narrower range of capabilities, as shown in Table 415 on page 855:

	Load Modules	**Program Objects**
Library type	PDS	PDSE, z/OS Unix HFS
Executable module	One-dimension; single AMODE, RMODE	Two-dimensions; multiple segments and A/RMODEs
Size limit	16MB	1GB
External symbols	8 characters	32K characters
Symbol types	SD, LD, ER, WX, PR	Same, plus ED
Additional module data	IDR only; no system support for access	Any data; Binder Programming Interfaces for access
Extensibility	Not possible	Open-ended architecture

Table 415. Comparing load modules and program objects

Exercises

38.8.1.(2)+ Assemble the little program in Figure 599 on page 849 with and then without the `GOFF` option, and compare their External Symbol Dictionaries. Which fields are significantly different?

38.8.2.(2) Compare the CESDs in Figure 583 on page 836 and Figure 595 on page 845 and explain the differences.

38.8.3.(1) The older Linkage Editor supported an extended form of overlay called "regions" that provided up to four distinct areas in which distinct overlays were supported. Try to find (and study) information about overlay regions.

38.9. Loading Saved Modules into Storage

The Program Loader brings load modules and program objects into storage, and relocates all address constants.

38.9.1. Loading Load Modules

As indicated in Figure 597 on page 847, a load module is a single unit of instructions, data, and uninitialized space. It has a single set of attributes: AMODE, RMODE, and reenterability (RENT) status, used by the Program Loader to load the module in the proper area of memory, and set the addressing mode before transferring control to the module's entry point.[268]

Adcon relocation is simple: the module's single load address is added to the text field at the offset given by the (linker-adjusted) Position address in the RLD item. This relocation is fast and efficient. It's important to remember that two stages of relocation are involved:

1. When the load module is created, the Linker (or Binder) relocates addresses relative to the zero module origin.
2. When the load module is brought into memory by the Program Loader, it relocates addresses relative to the module's load address.

---

[268] If the RENT option was specified during linking, the Program Loader will enable memory protection on the area where the module is loaded, so that any attempt to store in that area will cause a memory-protection program interruption.

To save having to read (and discard) the SYM, IDR, RLD, and CESD records, a copy of the first control record is saved in the PDS directory. It is used so the Program Loader can start reading the load module at the first text record. Each control record contains the length and the relative address of the following text record, and RLD information for adcons in that block of text.

The first block of text is loaded starting on a boundary aligned as specified by the SECTALGN option or by Linker and Binder control statements. If RMODE(ANY) was specified, the load module can be placed either above or below the 16MB "line".

38.9.2. Loading Program Objects

Program objects add flexibility by allowing you to arrange groups of Classes that are loaded into more than one area of memory. With System z, available memory is divided into three areas, as sketched in Figure 609, showing the three areas into which programs and data can be loaded.[269]

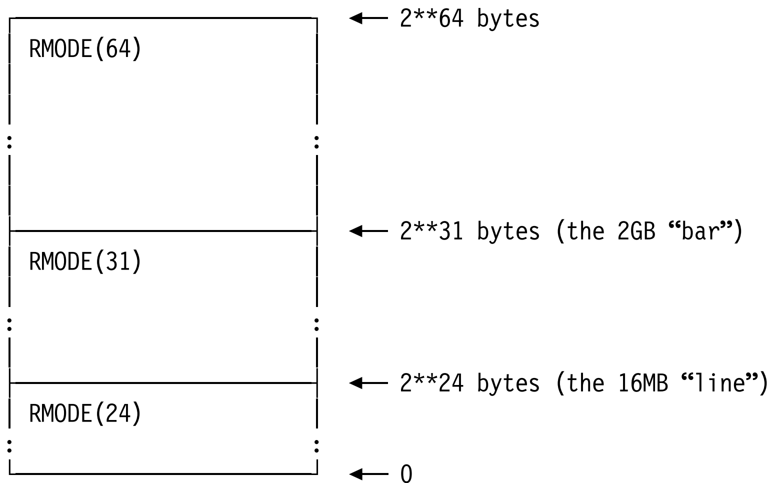


Figure 609. Sketch of virtual memory

The RMODE operand of the CATTR statement lets you indicate into which of these three areas you want the Class to be loaded by the Program Loader. Suppose the little program in Figure 603 on page 851 has been revised as in Figure 610, where we have simply added an RMODE(31) operand to the first CATTR statement for CLASS_Y:

```
SECT_A   CSect ,
         STM   14,12,12(13)    Appears in Class B_TEXT
CLASS_X  CATTR ,
Msg_1    DC    C'A message'    Appears in Class CLASS_X
SECT_B   CSect ,
         LM    14,12,12(13)    Appears in Class B_TEXT
CLASS_X  CATTR ,
Msg_2    DC    C'Good News'    Appears in Class CLASS_X
SECT_A   CSect ,
CLASS_Y  CATTR RMODE(31)       Specify RMODE(31) for CLASS_Y  ←—
Msg_3    DC    C'Better News'  Appears in Class CLASS_Y
SECT_B   CSect ,
CLASS_Y  CATTR ,
Msg_4    DC    C'Enough!'      Appears in Class CLASS_Y
```

Figure 610. Sample program defining two Sections and three Classes

[269] At the time of this writing, only data can be loaded into the area "above the bar".

If the GOFF object file from this assembly is linked into a program object, the Binder creates three *segments* from classes having the same attributes. The segments contain the items shown in Figure 604 on page 851, but now CLASS_Y has a different RMODE:

B_TEXT Residence mode 24 (the default)

CLASS_X Residence mode 24 (also the default)

CLASS_Y Residence mode 31 (as declared)

The Binder creates a program object with these segments; when the Program Loader brings the program into memory, the first two segments will be loaded "below the line" in the RMODE(24) area, and the CLASS_Y segment will be loaded "above the line, below the bar" in the RMODE(31) area.

To simplify loading, the Binder may combine segments with identical attributes into a single loading segment. In Figure 611, the **B_TEXT** and **CLASS_X** segments may be combined, so the loaded program could look somewhat like this:

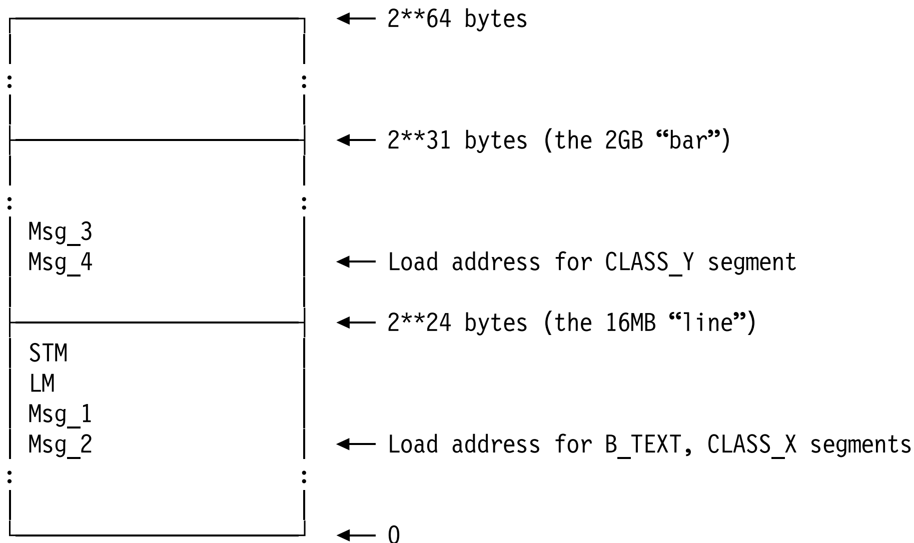


Figure 611. Sketch of classes in virtual memory

Loadable segments are loaded as *separately relocatable discontiguous* entities. Each is loaded as a single "block", similar to a load module, and inter-segment references are resolved correctly, even across different RMODEs.

The great value of loading different parts of a program into different areas of memory is that each segment may contain address constants referencing positions loaded into a *different* area! Unlike Load Modules that are loaded into only one area (and therefore can use address constants to refer only to positions within the loaded module), program objects can reference other areas.

In Section 38.10 we will see how to pass control among routines using different addressing modes, and how to reference different areas of memory.

Exercises

38.9.1.(2)+ Create a program with more than one control section, with references among the sections and their entry points. When the program is loaded into memory, dump its contents and verify that all address constants have been relocated correctly.

38.9.2.(1)+ Create a loadable module on your system and use whatever tools are available to display or dump its format prior to loading.

38.10. Changing Addressing Modes

Most application programs don't need to change addressing mode. They can access Operating System services that may change mode, but they restore your addressing mode on completion so the process is invisible to you.

These are typical situations where you might want or need to change addressing mode:

- Your application is loaded "above the line" and executes in 31-bit mode. It needs to call a program loaded below the line that runs in 24-bit mode.
- Your application is loaded below the line and runs in 24-bit mode, and needs to access data that resides above the line.
- Your application needs to access data above the 2GB "bar".

First, we describe five instructions you can use to change addressing mode, and one to test the current addressing mode. Of these instructions, BASSM and BSM are the most useful. Then we'll see how to use them.

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
0B	BSM	RR	Branch and Set Mode	0C	BASSM	RR	Branch and Save and Set Mode
010C	SAM24	E	Set Addressing Mode (24)	010D	SAM31	E	Set Addressing Mode (31)
010E	SAM64	E	Set Addressing Mode (64)	010B	TAM	E	Test Addressing Mode

Table 416. Instructions to change addressing mode

SAM24, SAM31, and SAM64 simply change the addressing mode to the indicated value, and TAM sets the Condition Code according to the current addressing mode:

AMODE	TAM CC
24	B'00'
31	B'01'
64	B'11'

Table 417. CC settings for TAM instruction

As noted in Section 20.2 on page 307, the two PSW bits that determine the addressing mode, the Extended addressing mode bit **E** and the Basic addressing mode bit **B**, are shown in Figure 612.

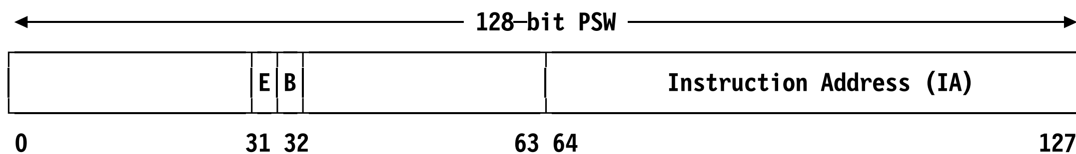


Figure 612. System z PSW showing addressing-mode bits

The meanings of the **E** and **B** bit settings are:

E	B	Addressing mode
0	0	24-bit mode
0	1	31-bit mode
1	0	Invalid combination
1	1	64-bit mode

Table 418. PSW addressing-mode bits

Note that the CC bit settings for TAM (in Table 417) are the same as the PSW **E** and **B** bit values (in Table 418).

The easiest way to change addressing modes is to execute one of the SAMxx instructions; this requires keeping track of the current and the new addressing modes in case you need to change back to the previous mode.

38.10.1. The BASSM Instruction

If your program (the caller) wants to call another program (the callee, or target program) in another addressing mode, use the BASSM instruction:

```
BASSM R1,R2
```

The important **b** and **e** bits in the R_1 and R_2 registers are shown in Figure 613.[270]

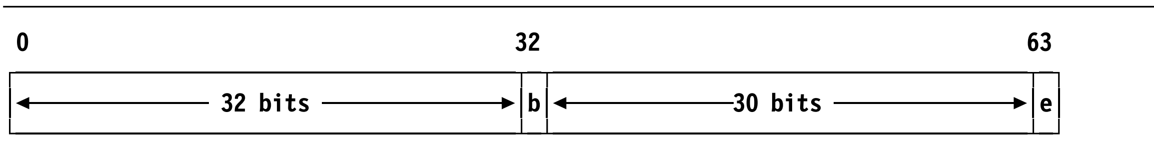


Figure 613. Important addressing mode bits for BASSM

When BASSM is executed, the CPU first completes the contents of GR R_1 or GG R_1, as shown in Figure 614. Thus, the R_1 register contains the return address, the address of the instruction following the BASSM, and bits indicating the addressing mode of the *calling* program. This lets the *called* program return to the caller in the addressing mode the caller used before the call.

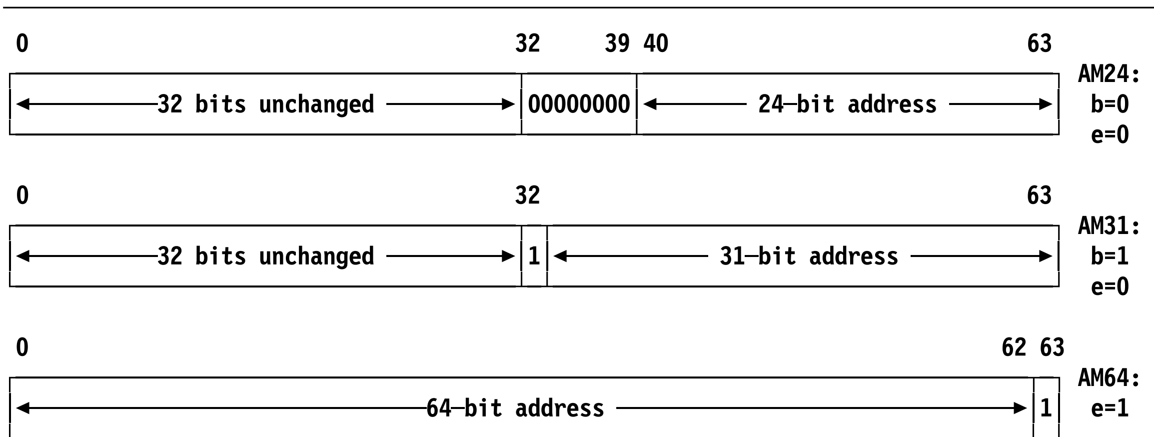


Figure 614. BASSM setting of first-operand register for 24-, 31-, and 64-bit addressing modes

If the R_2 digit is 0, the instruction is complete; no branch occurs, and the current addressing mode is unchanged.

If the R_2 digit is not 0, the CPU examines the **b** and **e** bits of GR R_2 to determine what change (if any) should be made to the addressing mode.[271]

1. If the **e** bit is zero (as you might expect for the *even* address of the callee), then the PSW addressing mode is set according to the value of the **b** bit:
   - If **b** = 0, the **E** and **B** bits in the PSW are set to `B'00'`, to set the new addressing mode to 24.
   - If **b** = 1, the **E** and **B** bits in the PSW are set to `B'01'`, to set the new addressing mode to 31.

---

[270] It's important to note the difference between the **E** and **B** bits in the PSW, and the **e** and **b** bits in a register.

[271] If you want to set $R_1 = R_2$ (a rare combination) consult the *z/Architecture Principles of Operation* first.

2. If the **e** bit of GR R_2 is one, the addressing mode **E** and **B** bits in the PSW are set to B'11', and the CPU will now execute in 64-bit addressing mode.

To complete the instruction, the CPU takes these additional steps:

1. If the new addressing mode is 24 or 31, the branch is taken and the next instruction will come from the 24- or 31-bit address in GR R_2.
2. If the new addressing mode is 64, the low-order bit of GG R_2 was set to one (so the branch address will appear to be odd!), but the CPU ignores the low-order one bit and takes the next instruction from the even address, without changing that low-order bit. (We'll see why this is done when we describe the BSM instruction.)

The new value for the PSW is computed before R_1 is changed.

Instruction fetching then continues at the target address in the new addressing mode.

A summary of the operation of BASSM is shown in Table 419.

BASSM	**to 24/31**	**to 64**
from 24/31	GR R_1: bits 0-31 unchanged bit 32 = **B** bit bits 33-63 = PSW IA PSW IA = Addr in GR R_2 (bit 63 = 0) PSW **E** bit = 0 PSW **B** bit = GR R_2 bit 32 Note: GR R_2 bit 63 = 0	same same same PSW IA = 64-bit addr in GR R_2 (bit 63=0) PSW **E** bit = 1 PSW **B** bit = 1 Note: GR R_2 bit 63 = 1
from 64	GR R_1 bits 0-62 = PSW IA (bit 63 = 1) PSW IA = Addr in GR R_2 (bit 63 = 0) PSW **E** bit = 0 PSW **B** bit = GR R_2 bit 32 Note: GR R_2 bit 63 = 0	GR R_1 bits 0-62 = PSW IA (bit 63 = 1) PSW IA = 64-bit addr in GR R_2 (bit 63=0) PSW **E** bit = 1 PSW **B** bit = 1 Note: GR R_2 bit 63 = 1

Table 419. BASSM actions summary

38.10.2. The BSM Instruction

The BSM instruction is used to restore the addressing mode of a calling program before returning. Its operation is summarized in Table 420; we'll use the notations in Table 417 on page 858 and Figure 613 on page 859 to refer to specific PSW and register bits.

Value	**R_1 operand**	**R_2 operand**
0	R_1 register unchanged	No branch, no change to addressing mode
≠0	• In AMODE 24 or 31, put the PSW **B** bit into the **b** bit of GR R_1; remainder of GR R_1 is unchanged. • In AMODE 64, insert a 1-bit into the **e** bit of GR R_1; remainder of GR R_1 is unchanged.	• If the **e** bit of GR R_2 is zero, set the PSW **E** bit to 0; the **b** bit of GR R_2 is put in the PSW **B** bit; AMODE is now 24 or 31. The 24- or 31-bit branch address replaces the PSW IA. • If the **e** bit of GR R_2 is one, set the PSW **EB** bits to B'11'; AMODE is now 64. The branch address in bits 0-62 of GR R_2 with a low-order zero bit appended replaces the PSW IA.

Table 420. Operation of BSM instruction

Of these combinations, the R_1 operand is almost always zero.

As noted above for BASSM, the new value for the PSW is computed before R_1 is changed.

A similar summary of the operation of BSM is shown Table 421 on page 861.

BSM	to 24/31	to 64
from 24/31	GR R_1: bits 0-31 unchanged bit 32 = **B** bit bits 33-63 unchanged PSW IA = 24-/31-bit addr in GR R_2 PSW **E** bit = 0 PSW **B** bit = GR R_2 bit 32 Note: GR R_2 bit 63 = 0	same same same PSW IA = 64-bit addr in GR R_2 (bit 63 = 0) PSW **E** bit = 1 PSW **B** bit = 1 Note: GR R_2 bit 63 = 1
from 64	GR R_1 bits 0-62 unchanged GR R_1 bit 63 = 1 PSW IA = 24-/31-bit addr in GR R_2 (bit 63 = 0) PSW **E** bit = 0 PSW **B** bit = GR R_2 bit 32 Note: GR R_2 bit 63 = 0	same same PSW IA = 64-bit addr in GR R_2 (bit 63 = 0) PSW **E** bit = 1 PSW **B** bit = 1 Note: GR R_2 bit 63 = 1

Table 421. BSM actions summary

38.10.3. Branch and Return With Addressing Mode Change

The BASSM and BSM instructions are quite similar to BASR and BR, respectively. Both are used for calling and returning from other routines. The major difference is that BASSM and BSM can change addressing modes during the call and the return.

Changes of addressing mode are usually used for calls among programs loaded into different areas of memory, that execute in different addressing modes. Such programs or memory areas may not be accessible in the current addressing mode, or must execute in a different mode.

1. Suppose your program **A** executes in AMode 31 and you must call a program **B** that executes in AMode 24 (and was therefore loaded below the 16MB "line").

```
L     15,=V(B)            Address of B (X'00xxxxxx')
BASSM 14,15               Call B
```

   B will be entered in 24-bit mode because bit 32 of GR15 is 0 (remember that the Program Loader sets the high-order 8 bits of a 4-byte adcon to zero if the target symbol is below 16MB). The **b** bit in GR14 will be set to 1 because the BASSM instruction was executed in AMode 31. **B** will then return with

```
BSM   0,14                Return from B to A
```

   and the addressing mode will be changed from 24- to 31-bit mode because the **b** bit in GR14 was 1.

2. Suppose your program **C** executes in AMode 24 and you must call a program **D** that executes in AMode 31. If **C** and **D** are in separate classes in a program object, then **D** could have been loaded above the 16MB "line". If, however, **C** and **D** are components of a load module, then the load module will be loaded below the 16MB "line".

```
EXTRN D                       D is an external routine
L     15,=A(D+X'80000000)     Address of D with bit 32 = 1
BASSM 14,15                   Call D
```

   D will be entered in 31-bit mode because bit 32 of GR15 is 1; we set the **b** bit of R_2 by adding X'80000000') to the address of **D**. (Remember that if **D** had been loaded below the 16MB "line", it will still be addressable in AMode 31.) The **b** bit in GR14 will be set to 0 because the BASSM instruction was executed in AMode 24. **D** will then return with

```
BSM   0,14                Return from D to C
```

   and the addressing mode will be changed from 31- to 24-bit mode because the **b** bit in GR14 was 0.

3. Suppose your program **E** executes in AMode 31 and you must refer to data named **F** that resides below the 16MB "line".

```
         L     15,=V(F)          Address of F (with bit 32 = 0)
```

You can now refer to the data at **F** because 31-bit mode lets you address anything below the 2GB "bar".

4. Suppose your program **G** executes in AMode 24 and you must refer to data named **H** that resides above the 16MB "line" but below the 2GB "bar".

```
         L     15,=V(H)          Address of H
         SAM31 ,                 Set addressing mode 31
```

You can now refer to the data at **H** because 31-bit mode lets you address anything below the 2GB "bar". If your program must later execute in 24-bit addressing mode, you will need to execute a SAM24 instruction.

The following Table 422 shows combinations of instructions that may safely be used by programs executing in the "From" addressing mode to call programs that execute in the "To" mode, and return to the caller restoring the caller's addressing mode.

From	*To* AM24		*To* AM31		*To* AM64	
	Call	**Return**	**Call**	**Return**	**Call**	**Return**
AM24	BAL BALR	BR				
AM24	BAS BASR BRAS BRASL	BR BSM	BASSM	BSM	BASSM	BSM
AM31	BASSM	BSM	BAL BALR BAS BASR BRAS BRASL	BR BSM	BASSM	BSM
AM64	BASSM	BSM	BASSM	BSM	BAL BALR BAS BASR BRAS BRASL	BR

Table 422. Instruction pairs for call/return with possible AMODE change

Note that the BAL instruction appears in the table only for branch to and and return from the *same* addressing mode. When your program is executing in addressing mode 24, BAL should be used *only* in combination with BR (and never with BSM) as the return instruction.[272] If BAL or BALR is the target of an EX or EXRL instruction, only BR can be used to return to the caller. (See Exercise 38.10.7.)

Table 423 on page 863 shows ways you can do internal subroutine calls that change addressing modes within a single assembly.

[272] The official term describing BAL and BALR is that their use is "deprecated". This means that if you get in trouble using them, you can't say you weren't warned.

Caller	Subroutine Entry

`* To 24-bit mode from any mode`
`        LARL  15,Code24`
`        BASSM 14,15`
	`* Entry to 24-bit mode`
	`Code24  DC    0H     **Must be below the line**`
	`        - - -`
	`        BSM   0,14   Return to caller`

`* To 31-bit mode from any mode`
`        LARL  15,Code31`
`        OILH  15,X'8000'`
`        BASSM 14,15`
	`* Entry to 31-bit mode`
	`Code31  DC    0H     **Above or below the line**`
	`        - - -`
	`        BSM   0,14   Return to caller`

`* To 64-bit mode from any mode`
`        XGR   15,15  (For 24 or 31 to 64)`
`        LARL  15,Code64`
`        OILL  15,X'0001'`
`        BASSM 14,15`
	`* Entry to 64-bit mode`
	`Code64  DC    0H     **Anywhere in memory**`
	`        - - -`
	`        BSM   0,14   Return to caller`

Table 423. Calling among addressing modes within an assembly

When a program executes in 64-bit addressing mode, it can address any byte with address between 0 and $2^{64}-1$. However, on z/OS systems the addresses between 2^{31} and $2^{32}-1$ are not made available; this area is sometimes called the “Blackout Area”, as sketched in Figure 615.

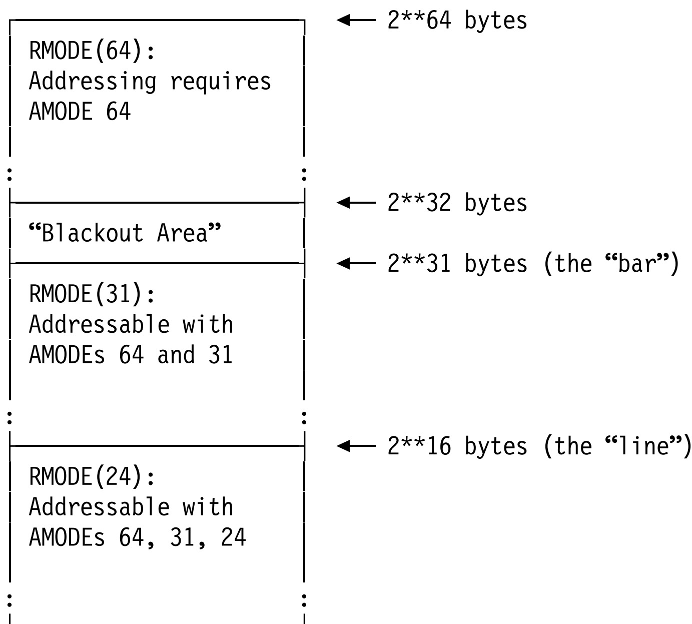


Figure 615. Sketch of residence and addressing modes

This was done because many existing programs used the convention shown in Section 37.3.1 on page 767 that set the high-order bit of a 4-byte address to 1 to indicate the last argument address in a list. If such an address is used in 64-bit addressing mode it will (accidentally) try to refer to an address in the Blackout Area, rather than the intended address below the 2GB “bar”.

38.10.4. Load Logical Thirty-One Bits Instructions

To help avoid such problems we can use the LLGT and LLGTR instructions:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
E317	LLGT	RXY	Load Logical Thirty One Bits	B917	LLGTR	RRE	Load Logical Thirty One Bits

Table 424. LLGT and LLGTR instructions

These two instructions load a 32-bit word into the low-order 32 bits of the GR R_1 register, set bit 32 to 0 (the bit that could potentially be 1) and set the remaining 32 high-order bits of GR R_1 to

zero. This means that the resulting address can safely be used by a program executing in 64-bit mode to refer to addresses below the 2GB “bar”. For example:

```
         LLGT  4,=X'FEDCBA98'     c(GG4=X'00000000 7EDCBA98')
```

Note that bit 32 of GG4 has been set to zero.

The example in Figure 616 shows why this can be important. Suppose your program runs in AMode 31, and then changes to AMode 64.

```
Prog31   CSect ,
Prog31   AMODE 31                 Execute in AMode 31
         BASR  12,0               Establish base register
         USING *,12               Establish addressability
         - - -
         SAM64 ,                  Change to AMode 64
         - - -
Move     MVC   A,B                Move some data
         - - -
A        DS    CL10               Target field
B        DC    C'A Message!'      Source field
```

Figure 616. Example showing why LLGT/LLGTR are necessary

The MVC instruction named **Move** may cause a protection exception for either of two reasons. The BASR instruction, because it is executed in AMode 31, sets bit 32 of GG12 (the **b** bit) to 1. When the Effective Addresses of the MVC operands **A** and **B** are generated, the presence of the 1-bit means that

- if bits 0-31 of GG12 are zero, the Effective Address of **A** lies in the “Blackout Area”;
- otherwise, if bits 0-31 of GG12 are not zero, the Effective Address of **A** lies somewhere above 2^{32}, in an area very probably not accessible to your program.

To avoid both forms of this problem, insert a LLGTR instruction:

```
         LLGTR 12,12              Force bits 0-32 of GG12 to zero
         SAM64 ,                  Change to AMode 64
         - - -
Move     MVC   A,B                Move some data
```

The following example shows other reasons why LLGTR can be important. Suppose you have written a program named **Code64** that executes in AMode 64, and is called by another program that executes in 24- or 31-bit addressing mode. The caller provides a standard Format-0 save area addressed by R13, and a pointer in R1 to two word addresses referencing to arguments. The call is made using a **BASSM 14,15** instruction. We assume that the calling sequence is like this:

```
         LA    13,SaveArea        Point to local Format-0 save area
         LA    1,ArgList          Point to argument addresses
         LLGT  15,=V(Code64)      Get address of called routine
         AHI   15,1               Set 64-bit mode indicator bit
         BASSM 14,15              Branch and set mode 64
         - - -
ArgList  DC    A(Arg1,Arg2)       Argument addresses
```

The called routine in Figure 617 on page 865 could look like this:

```
Code64   CSect ,
Code64   AMode 64                64-bit addressing mode
         Using *,15              Local base register
         NILL  15,X'FFFE'        Turn off low-order bit of GG15
         LLGTR 13,13             Clean address of caller's area in GG13
         STMH  14,12,HighRegs    Save high halves of registers locally
         STM   14,12,12(13)      Store low halves of R14-R12
         LLGTR 1,1               Make a valid arg-list pointer
         LM    1,2,0(1)          Load argument addresses
         LLGTR 1,1               Usable pointer to argument 1
         LLGTR 2,2               Usable pointer to argument 2
         - - -
Return   LMD   14,12,HighRegs,12(13)  Restore both halves of registers
         BSM   0,14              Return to caller
HighRegs DS    15F               Save area for high halves of registers
```

Figure 617. Example showing why LLGTR is important

- Without the first LLGTR instruction, any nonzero bits in positions 0-33 of GG13 could cause a protection exception. The same applies to the LLGTR instructions referencing GGR1 and GGR2.
- We assume that we can return to the caller with the high-order 33 bits of GG13 set to zero. This should not cause a problem, because the address of the caller-provided save area is in his program.
- On return, we did not restore the low-order 1-bit in GG15 because it was set only for the use of BASSM.

> **Be Careful!**
>
> It is important that calling and called routines agree on their respective addressing modes, linkage conventions, register-preservation expectations, and argument-passing conventions.

Exercises

38.10.1.(1)+ When a program is loaded into memory, what is the maximum allowed difference between the addresses of BRAS and BRASL instructions and their targets?

38.10.2.(1) Show which PSW addressing mode bits (Figure 612 on page 858 may help) are set to what values by each of the SAMxx instructions in Table 416 on page 858.

38.10.3.(1)+ Show how the CC settings in Table 417 on page 858 correspond to the settings of the E and B addressing-mode bits in the PSW, as illustrated in Figure 612 on page 858.

38.10.4.(2)+ Describe the differences among `BALR r,0` and `BASR r,0` and `BASSM r,0`.

38.10.5.(1)+ The *z/Architecture Principles of Operation* states that if the current addressing mode is 24 or 31, the BSM instruction can be used to return from a program that was entered using a BAS, BASR, BRAS, and BRASL instruction. Explain why this is so.

38.10.6.(1) Can `BASSM 0,0` or `BSM 0,0` be used as a no-operation instruction, like `NOPR 0`?

38.10.7.(2)+ The paragraph after Table 422 on page 862 states that BAL should never be used with BSM as its return instruction when your program executes in AMODE 24. Why? Why is AMODE 31 not included in this warning?

38.10.8.(2) In Table 422 on page 862, why not use BASSM and BSM for a branch and return within the same addressing mode?

38.10.9.(4) In Figure 617, it was assumed that the calling and called routines executed in the same addressing mode. Write instructions to handle the case(s) where the calling and called routines *might* execute in different addressing modes.

38.10.10.(2)+ In example 2 on page 861, the text says that routine **D** "could have been loaded above the 16MB 'line'." What determines where it is loaded?

38.11. Summary

This chapter has covered a wide variety of topics. Key suggestions include:

1. Put all constants that can't be replaced by immediate operands in one LOCTR group. It will probably need addressability.
2. Put all read/write data areas in one LOCTR group. It will probably need addressability.
3. Arrange for all instructions to be in one LOCTR group. Use relative branch, immediate, and long-displacement instructions wherever possible. Ideally, this LOCTR group will not need addressability.
4. Avoid using BAL/BALR except in applications with no changes of addressing mode.

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
BASSM	0C
BSM	0B
LLGT	E317

Mnemonic	Opcode
LLGTR	B917
SAM24	010C
SAM31	010D

Mnemonic	Opcode
SAM64	010E
TAM	010B

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
010B	TAM
010C	SAM24
010D	SAM31

Opcode	Mnemonic
010E	SAM64
0B	BSM
0C	BASSM

Opcode	Mnemonic
B917	LLGTR
E317	LLGT

Terms and Definitions

A-type address constant
A field containing an absolute, relocatable, or complex relocatable expression. Absolute expressions are resolved by the Assembler, the others during linking and loading.

address constant ("adcon")
A field within a control section into which a value (typically, an address) is placed during program binding, relocation, and/or loading.

AMODE
Addressing mode: one of 24, 31, or 64.

Class
: (1) A cross-section of program object data with uniform format, content, function, and behavioral attributes. (2) A component of a program object with specified loading properties, containing elements supplied by sections. Loadable classes are independently relocatable. Indicated in the External Symbol Directory listing with type ED.

Common
: A CSECT having length and alignment attributes (but no text) for which space is reserved in the load module or Program Object.

CXD-type address constant
: A word holding the length (not the address) of the virtual area created at link time from all the Dummy External Sections (or PseudoRegisters) in the complete program.

element
: A component of a program object Class, defined by the combination of its section name and its Class name. The smallest indivisible and separately relocatable portion of a program object.

executable control section
: A control section containing machine language instructions or data, defined by CSECT, RSECT, or START instructions.

External Symbol Dictionary
: The set of external symbols created by an assembly. They are displayed in the Assembler's listing and are encoded in the ESD records of the generated object module.

linking loader
: Links and places modules into storage with linking, immediately prior to program execution.

load module (LM)
: The original form of MVS executable, stored in a Partitioned Data Set (PDS) program library in “record format”.

object module
: Records containing the external symbols, machine language text, and relocation dictionary information required for program linking, in either the older 80-byte card image “OBJ” format or in the newer GOFF format.

program linking
: The process of resolving external names into offsets or addresses; combining multiple input name spaces into a composite output name space.

program object (PO)
: A newer form of executable on z/OS, stored in a PDSE (Partitioned Data Set Extended) program library.

PseudoRegister (PR), External Dummy Section (XD)
: A PseudoRegister or external data item having length and alignment attributes. Space in the loaded module is reserved for Common control sections; space for External Dummy Sections must be obtained at execution time.

Q-type address constant
: A field containing the offset (not the address) of an External Dummy Section (or PseudoRegister) from the start of a virtual area mapped at link time and allocated at execution time.

reference control section
: A control section containing no machine language instructions or data, defined by DSECT, COM, or DXD instructions.

relocate
: Assign actual-storage or module-origin-relative addresses to address constants.

relocating loader
: Places modules into storage *and* adjusts addresses to their correct “final” value.

relocation
: The load-time conversion of address constants from module or Class displacements to virtual addresses. The assignment of actual or module-origin-relative addresses to address constants.

Relocation Dictionary
A summary of each relocatable address constant in an assembly, displayed in the Assembler's listing and encoded in the RLD records of the generated object module.

RMODE
Residence mode, an indication of the desired placement in memory of a Control Section or Class.

section
(1) A generic term for control section, dummy section, common section, etc.. A collection of items that must be bound or relocated as an indivisible unit. (2) A collection of *elements* belonging to specified *Classes* in a program object. Elements defined by a section are added or deleted as a group. (A program object section is not the same as a control section.)

segment
A component of a program object containing classes with the same properties such as RMODE and loadability.

text
(1) The instructions and data generated by an assembly, encoded in the TXT records of an object module. (2) A program object Class attribute indicating that locations within the class may contain and/or be the target of address constants.

V-type address constant
A field containing the address of an external symbol, resolved during linking and loading.

Programming Problems

Problem 38.1.(2) Use the external subroutines you wrote in solving Exercises 38.5.15 and 38.5.16 to create a complete program calling the subroutines. Verify your results by converting word integers to hexadecimal floating-point and then back to word integers; display the original and final integer values and the hexadecimal floating-point intermediate value. Be sure to test values such as

```
DC    F'-0'
DC    F'2147483647'
DC    F'-2147483648'
```

Problem 38.2.(4)+ Days of the week can be calculated using *Zeller's Congruence*:[273]

```
D_W = (D_M + ((M+1)*13)/5) + Y + Y/4 + 6*(Y/100) + Y/400) (mod 7)
```

where D_M is the day of the month, M is the month, and Y is the year. D_W is the day of the week, starting with 0=Saturday through 7=Friday. All divisions (except the one calculating the day of the week "(mod 7)") discard remainders.

Write a separately assembled subroutine **Zeller** to accept three integer arguments Y, M, and D_M and return D_W in GR0. Then, write a calling program that reads records with the three argument values right-justified in columns 1-10, 11-20, and 21-30. For example, some input records might be

```
2009         6        30
2000         1         1
1900         1         1
```

Create several test cases that display the day of the week, the month, day of the month, and the year. Even better, provide the *names* of the day and month.

Problem 38.3.(2)+ Using the definitions of ulp(x) in Section 32.8 on page 580, and with the assumptions in Exercises 33.19.3, 33.19.5, and 33.19.7, write three callable subroutines using standard calling conventions that are all in a single control section named **HexUlp**. Your CSECT should have three entry points: **HxUlpE**, **HxUlpD**, **HxUlpL**, having a short, long, and extended precision hexadecimal floating-point argument respectively.

[273] Due to Christian Zeller, in 1882-85. The formula is valid only for the Gregorian calendar.

Each entry should calculate the ulp of its argument and return the result in floating-point register 0 (for **HxUlpE** and **HxUlpD**) and in floating-point registers 0 and 2 for **HxUlpL**.

Your routines should preserve the contents of general registers 2-15 and floating-point registers 8-15 if they are modified in any way. Assume that the values of the arguments are large enough that no exponent underflows will occur in calculating each ulp.

Problem 38.4.(3) As you did in Problem 38.3, write a program in a CSECT named **BinUlp** with three similar entry points that will evaluate an ulp of a short, long, and extended precision binary floating-point argument.

Problem 38.5.(2) Write a callable routine named **ConvI** with two arguments: a 32-bit binary integer, and a 12-byte character string. Convert the binary integer to characters, right-adjusted in the string, and preceded by a minus sign if the integer is negative. Insignificant leading zeros should be omitted, and at least one digit must be produced. Be sure to test values like

```
DC    F'0'
DC    F'-1'
DC    F'2147483647'
DC    F'-2147483648'
```

Problem 38.6.(2) Write a callable routine named **ConvZ** with two arguments: a 32-bit word and an 8-byte character string. Convert the word to 8 EBCDIC characters representing the hexadecimal representation of the word.

Problem 38.7.(4)+ Write a program to display the contents of an 80-byte “OBJ” object module in hexadecimal.

Problem 38.8.(2)+ Write at least two separately-assembled routines referring to DXD items, some of which have identical names. Include Q-type address constants of lengths 2 to 4 referring to each item, and at least one CXD-type item. Link the routines together and execute the linked program.

Examine (and print) the contents of all the Q-cons and CXD constants to understand how the Linker allocated the entries in the External Dummy Section.

Problem 38.9.(1) Write a short program to illustrate the settings of the R_1 register when you execute BASSM and BASR instructions with $R_2 = 0$, in each of the three addressing modes. (Be careful if you use the PRINTOUT macro to print the results, because it executes only in AMODE 24.)

Problem 38.10.(2) Write and test a subroutine named **HexExp** that will evaluate and return in GR0 the exponent of a hexadecimal floating-point argument passed using standard linkage conventions. (You need not restore the contents of GR1.)

Chapter XI: Dummy Sections, Enhanced USINGs, and Data Structures

```
XX        XX  IIIIIIIIII
XX        XX  IIIIIIIIII
 XX      XX       II
  XX    XX        II
   XX  XX         II
    XXXX          II
    XXXX          II
   XX  XX         II
  XX    XX        II
 XX      XX       II
XX        XX  IIIIIIIIII
XX        XX  IIIIIIIIII
```

This chapter describes some basic techniques for arranging and describing data items in memory, including arrays, stacks, linked lists, and hash tables.

- Section 39 introduces the Dummy Control Section (DSECT) for mapping data objects, and then reviews the ordinary USING statement before introducing the enhanced Labeled USING, Dependent USING, and Labeled Dependent USING assembler instruction statements.
- Section 40 describes some basic techniques for arranging data items in memory, including arrays, stacks, linked lists, and hash tables.

39. Dummy Control Sections and Enhanced USING Statements

```
 3333333333     9999999999
333333333333   999999999999
33         33  99         99
           33  99         99
           33  99         99
        3333   999999999999
        3333   999999999999
           33             99
           33             99
33         33  99         99
333333333333   999999999999
 3333333333     9999999999
```

This section describes two powerful programming aids: Dummy Control Sections and enhanced USING statements.

Dummy Control Sections are powerful tools for symbolic description of any area of memory. They are especially valuable when programs work with shared or multiple instances of data structures such as records.

Any addressing method should provide as many of the following benefits as possible:

1. Coding should be simple, clear, understandable, and efficient. These help with simplicity, readability, and maintainability.
2. All instructions should use fully symbolic references. These help with readability and maintainability.
3. Base registers and displacements should always be automatically assigned by the assembler from information provided in USING statements, and never as constants or by manual calculations. These also help with quality, readability, and maintainability.

Ordinary USINGs can easily fail in one or more of these respects, as some of the following illustrations will show; we will see how the new USING statements can avoid most of them.

39.1. Dummy Control Sections

In Sections 6.4 and 6.5 we mentioned that the START and CSECT assembler instruction statements initiate a control section containing instructions and data.[274] A Dummy Control Section (or DSECT) is a control section like a control section initiated by CSECT or START, with these similarities and differences:

- A DSECT never generates any machine language object code, even if it contains instructions and constants.

[274] The Assembler defines two types of control section: (a) an “executable” control section containing machine-language instructions and data (even though it might contain only data and no executable instructions, or even nothing at all!), and (b) a “reference” control section, most often a DSECT. (A reference control section can also be a COM section or an external dummy section, described in Section 38.4.)

- A DSECT has its own relocation attribute (RA). Because it generates nothing that will appear in the object module, its RA starts at (unsigned) X'FFFFFFFF' and counts down for each new DSECT. (You may remember from Section 38 that external symbols have relocation attributes defined by their ESDIDs, which count up from 1.)
- The Location Counter (LC) for a DSECT always starts at zero, which is treated as a doubleword boundary.[275] Normal alignment rules apply to each statement in a DSECT.

A DSECT is a *template* or *mapping* of an area of memory, and is used to make symbolic references.[276] To illustrate, suppose we declare a DSECT named **DummyS**:

```
DummyS   DSect ,                 Declare dummy control section DummyS
A        DS    F                 Fullword named A
B        DS    H                 Halfword named B
C        DS    CL25              25-byte character string named C
D        DS    D                 Doubleword named D
```

Figure 618. Example of a dummy control section

The symbol table entries for the four symbols might look like this:

Symbol	Location	RA
A	X'000000'	X'FF'
B	X'000004'	X'FF'
C	X'000006'	X'FF'
D	X'000020'	X'FF'

Table 425. Symbol table entries for DSECT symbols

The Relocation Attribute of each symbol in the DSECT is the same as the RA of its "owning" control section.

Now, suppose we can address a work area named **WorkA**. We can use symbols to refer to fields in the work area, as in Figure 619:

```
Sample   Start 0
         BASR  12,0              Establish a base register
         Using *,12              Provide addressability
         - - -
         LA    7,WorkA           c(GR7) = Address of WorkA
         Using DummyS,7          Provide symbolic mapping of WorkA
         L     3,A               Load a fullword from WorkA
         AH    3,B               Add a halfword from WorkA
         CLC   C,0(12)           Compare 25 bytes of WorkA
         JE    ElseWhere         Do something about the comparison
         LM    0,1,D             Load two words from doubleword
         - - -
WorkA    DS    0D,XL84           Work area
```

Figure 619. Example using a dummy control section

The instructions referencing symbols defined in the DSECT are generated using the normal resolution rules for implied addresses. With the same notation as in Sections 10.8-10.10, the USING Table would look like Figure 620:

[275] If you need to include items in a DSECT that require quadword alignment, specify the SECTALGN(16) option for the assembly.

[276] You can think of a DSECT as though it's a transparent overlay that lets you make symbolic references to fields anywhere in memory. (This description is attributed to Jim Morrison.)

basereg	base location	RA
12	00000002	01
7	00000000	FF

← Relocation ID for the DSECT

Figure 620. USING Table with two entries, one for a dummy section

References to the symbols defined in the DSECT can be resolved only with base register GR7, because the RA of the implied addresses in Figure 619 must match the RA of a base address in the USING Table. The generated instructions are shown in Figure 621:

```
Loc     Object Code             Statement
....    5830 7000               L     3,A
....    4A30 7004               AH    3,B
....    D518 7006 C000          CLC   C,0(12)
....    A784 ....               JE    ElseWhere
....    9801 7020               LM    0,1,D
```

Figure 621. Object code from references to a dummy control section

The symbol C is at location X'000006' in the DSECT; the second USING statement tells the Assembler that the base address is location X'000000', so the addressing halfword is X'7006'.

Now, suppose we refer to a different work area, so that the address in GR7 is different:

```
         - - -
         LA    7,WorkA+43          c(GR7) = A(WorkA)
         Using DummyS,7            Provide symbolic mapping of WorkA
         L     3,A                 Load a fullword from WorkA
         AH    3,B                 Add a halfword from WorkA
         CLC   C,0(12)             Compare 25 bytes of WorkA
         JE    ElseWhere           Do something about the comparison
         LM    0,1,D               Load two words from doubleword
         - - -
WorkA    DS    XL84                Work area
```

Figure 622. Example using a dummy control section

The generated object code is identical to that in Figure 621, because exactly the same base expression and register are declared in the USING statement. In resolving the implied addresses, the Assembler doesn't know that the address in GR7 may not be aligned on proper boundaries; it only knows about alignments within the DSECT.

The examples in Figures 619 and 622 show how the DSECT template can be “overlaid” on any area of storage to provide symbolic references (the symbols **A**, **B**, **C**, and **D** to fields in the storage area.

Because a DSECT generates no machine language object code, we could have defined the DSECT in Figure 618 with DC (rather than DS) statements, with exactly the same results:

```
DummyS   DSect ,                   Declare dummy control section DummyS
A        DC    F'0'                Fullword named A
B        DC    H'-45'              Halfword named B
C        DC    CL25'String'        25-byte character string named C
D        DC    D'-1.732'           Doubleword named D
```

The Length Attribute of a symbol naming a DSECT is 1, and *not* the total length of the DSECT itself.

Exercises

39.1.1.(1) In this DSECT, show the values of each symbol.

```
D39_1_1  DSect ,
A        DS    CL9
B        DS    F
C        DS    X
D        DS    H
E        DS    D
F        Equ   *-D39_1_1
```

39.1.2.(1)+ Figure 57 on page 161 shows statements describing a typical Assembler Language statement. Create a DSECT named ALStmt with those statements.

39.1.3.(2)+ In the DSECT you wrote in Exercise 39.1.2, what are the values and length attributes of all symbols?

39.2. Multiple Data Structures

You will often find that your program must work with more than one copy of an identical data structure. For example, suppose your program must read a data record from an old master file and update it in two ways:

- Copy the **Old** record to a **New** copy.
- Put the current processing date in the **Old** record and write it to an "archive" file for backup (and auditing) purposes.
- Make updates to the **New** record and write it to a master file.

In outline form, the main steps of your program might look like this:

```
BAS   14,Read_Old_Master      Read a record for processing
BAS   14,Copy_Record          Copy the old record to new area
BAS   14,Do_Old_Date          Put date stamp in old record
BAS   14,Write_Archive_Record Write old record to archive
BAS   14,Update_New_Record    Make updates to new record
BAS   14,Write_New_Master     Write new record to master file
```

When you work with more than one copy of a data structure, your programs are simpler when the structures are described by a DSECT. Thus, each subroutine needs to know the structure and length of the record. Suppose the two record descriptions had been written as two separate groups of statements:

```
New Record Description                       | Old Record Description
---------------------------------------------|-----------------------
NewRec   DS    0D                            |OldRec   DS    0D
NewType  DS    CL10   Record type            |OldType  DS    CL10
NewID    DS    CL4    Record ID              |OldID    DS    CL4
NewName  DS    CL40   Name                   |OldName  DS    CL40
NewAddr  DS    CL66   Address                |OldAddr  DS    CL66
NewPhone DS    CL12   Phone number           |NewPhone DS    CL12
         - - -        etc.                   |         - - -
         - - -        etc.                   |         - - -
NewYear  DS    F      Processing year        |OldYear  DS    F
NewDay   DS    F      Day of year            |OldDay   DS    F
         - - -        etc.                   |         - - -
```

Figure 623. A poor method for describing two instances of a record

and so forth for many fields. Any error in keeping the **New** and **Old** descriptions exactly the same could lead to serious errors if fields appeared at different offsets.

It is *much* better to describe the record with a DSECT:

```
Record   DSect ,                 Record description
RecType  DS    CL10              Record type
RecID    DS    CL4               Record ID
RecName  DS    CL40              Name
RecAddr  DS    CL66              Address
RecPhone DS    CL12              Phone number
         - - -                   etc.
         - - -                   etc.
RecYear  DS    F                 Processing year
RecDay   DS    F                 Processing day of year
         - - -                   etc.
RecLen   Equ   *-Record          Record length
```

Figure 624. A better record description with a DSECT

and then use the DSECT in each subroutine to process the data. (Section 39.4 describes Labeled USINGs, which make it even easier to handle multiple instances of a data structure.)

Advice

If your program references more than a single copy of a record or structure, or if a record or structure is shared by more than one program, *all* references should (must!) use the same DSect mapping. It's best to write only a *single* description of any data structure.

Your program can contain a mixture of both "regular" control sections and dummy control sections. Standard programming practice usually groups all DSECT definitions either at the very start or the very end of your program.

Exercises

39.2.1.(2)+ Suppose you want to refer to the **Record** structure in Figure 624 as a string of **RecLen** bytes. Define a symbol **RecBase** at offset zero in the DSECT having Length Attribute **RecLen**.

39.2.2.(1)+ Define a storage area named **NewRec** described by the **Record** DSECT in Figure 624 to start on a doubleword boundary, and have the correct length.

39.3. Shortcomings of Ordinary USING Statements

By "ordinary USING" we mean the USING statement we've seen many times previously, written as in Figure 625:

```
USING base_location,register(s)   Ordinary USING statement
```

Figure 625. Ordinary USING statement syntax

The other three forms of USING statement we'll discuss are variations on this basic syntax.

We will use the DSECT describing a typical data record shown in Figure 624 to illustrate several methods for copying the **RecID** field from the old record to the new. Each method has shortcomings; we'll see in Section 39.4 that these are easily overcome with Labeled USING statements.

Suppose our program refers to the new and old and instances of the **Record** DSECT, and that we must move the **RecID** field from the **Old** instance of the record to the **New** instance, as sketched in Figure 626:

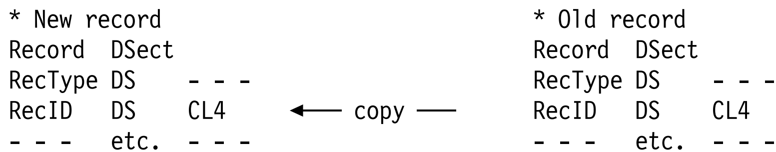


Figure 626. Copying a field from Old record to New

First, we will see examples of five ways we might move the data, specifying only ordinary USING statements:

- Example 1: Incorrect addressing with ordinary USINGs.
- Example 2: Ordinary USINGs, with manually-specified displacements.
- Example 3: Unusual ordinary USINGs, with manually-specified displacements.
- Example 4: Ordinary USINGs and an intermediate temporary variable.
- Example 5: Duplicated (but differently-named) copies of the DSECT.

For each example, we will assume that

- GR5 and GR7 contain the addresses of the **New** and **Old** record instances respectively, and
- a single MVC instruction is the desired efficient solution.

39.3.1. Ordinary USINGs

We will illustrate several approaches to managing the two copies of the **Record** DSECT with ordinary USING statements. Some of them are clearly incorrect; they are included to show how (apparently) obvious and simple solutions can lead to unexpected pitfalls.

Example 1 — Obviously Incorrect Usage: Suppose we wrote either of the two following sequences of statements:

```
Using Record,5                    Using Record,7
Using Record,7          or        Using Record,5
MVC   RecID,RecID                 MVC   RecID,RecID
```

We want the Assembler to generate the machine language instruction `X'D203 500A 700A'`.

Both sequences fail because only GR7 will be used to address the fields of the **Record** DSECT. (If two registers are based on the same location, the assembler chooses the higher-numbered register for base-displacement resolutions.) Thus, the generated machine language instruction is `X'D203 700A 700A'`, and the MVC instructions will move the **RecID** of the **Old** record onto itself, producing no result whatever![277]

The defects of this first technique are:

- incorrect code.

Example 2 — Correct (But Poor) Usage with Manually-Specified Displacements and Registers: Suppose now that we now rewrite these simple statements to avoid the previous problems, by specifying the displacement and base to be used:

```
Using Record,5              Map New instance of Record
MVC   RecID,RecID-Record(7) Move from Old to New
```

This sequence has the disadvantage that the displacement and base register are assigned by the programmer, rather than by the assembler. If there is ever a need to reassign base registers (so that GR7 is given a different use), all references to GR7 must be located and inspected to see if they need changing.

[277] The Assembler can provide a diagnostic message warning about the fact that GR5 has been “nullified” as a base register to alert you to this situation.

In summary, the defects of this technique are

- more complex coding;
- more difficult maintenance.

Having seen that you can explicitly assign the base and displacement this way, you might be tempted to use the same technique for the *first* operand of the MVC instruction:

```
Using Record,7               Map Old instance of Record
MVC   RecID-Record(5),RecID  Move from Old to New
```

Figure 627. Incorrect addressing with ordinary USING

While this is syntactically correct it will undoubtedly be wrong. The syntax rules of the Assembler Language state that if the first operand of an SS-type instruction is written in the form expr1(expr2), then expr1 provides the implied address and expr2 provides the operand's explicit Length Expression.

The more serious flaw is that because expr1 is absolute, the first operand will be resolved with base register zero, and therefore refer to the low-addressed end of storage, and may cause an interruption during program execution![278] (See Exercise 39.3.2.)

Consider how much more difficult this problem would have been to find if you had used a common register notation:

```
USING Record,R7              Map Old Record with GR7
MVC   RecID-Record(R5),RecID Move Old to New (based on GR5)
```

The symbol R5 in the MVC statement could lead many readers to believe that it was a correct register reference, while in fact it is the *Length* Expression! It will be a rare coincidence if the implied length of the RecID field is the same as the value of the symbol R5, so this might be "correct" by chance (but almost always incorrect) despite the lack of any diagnostics.

The correct form requires a comma in the first operand to indicate that its implied length should be used:

```
USING Record,7               Map Old instance of Record
MVC   RecID-Record(,5),RecID Move from Old to New
```

Figure 628. Correct but awkward addressing with ordinary USING

This requires remembering an obscure Assembler Language rule that may not be obvious to everyone. (See Exercise 39.3.3.)

Another potential trap in manually assigning registers is that USINGs may be in effect for both the **Old** and **New** register numbers, in such a way that a statement may assemble correctly but its operand(s) may be resolved with respect to the wrong register.

Example 2a — A Digression about Devious Programming: To avoid these syntactic difficulties, a clever programmer might observe that a manually-calculated displacement can be resolved without having to specify a base register explicitly by specifying a zero base address and the desired register:

```
USING Record,5               Map new instance of Record
USING 0,7                    Map old instance of Record (??)
MVC   RecID,RecID-Record     Move from Old to New
```

and the MVC instruction will now resolve correctly.[279]

However, if you forget to DROP register 0, later statements that depend on absolute expressions resolving with register 0 may not give the correct object code:

[278] The Assembler will try to diagnose such low-storage references if you specify the FLAG(PAGE0) option.

[279] The Assembler will tell you that your USING with absolute base address zero overlaps with its implicit USING 0,0.

```
         - - -                More statements (forgetting to drop R7)
         LA    1,100          Resolved by GR7! (X'41107064')
```

Clever programming has its limits.[280]

Example 3 — Problems with Manual Assignment: Suppose the data structure mapped by the **Record** DSECT grows to be larger than 4096 bytes. You would establish two base registers to map each of the two instances:

```
         LAY   6,4096(0,5)         Increment GR5 by 4096 into GR6 ...
         USING Record,5,6          Map New instance of Record
         LAY   8,4096(0,7)         Increment GR7 by 4096 into GR8 ...
*        USING Record,7,8          Implicit map of Old instance of A
```

Then, if you write

```
         MVC   RecID,RecID-Record(7)   Is (RecID-Record) now > 4095?
```

The correctness of the second operand depends on whether the manually-assigned displacement RecID−**Record** is *less* than 4095. If not, the displacement will be too large, and the manually-assigned register (7) will be incorrect. Thus, you might have to write something like this:

```
         MVC   RecID,RecID-Record-4096(8)   If (RecID-Record) > 4095
```

Figure 629. Manual coding of base and displacement for a large DSECT

This is obviously error-prone, since it depends on the current size of the **Record** DSECT and the known offset of the RecID field; both could change as the program evolves.

In summary, the defects of the techniques in Examples 2 and 3 are

- greater likelihood of undetected error;
- deeper understanding required of language details;
- more complex coding;
- more difficult maintenance.

Example 4 — Correct (But Still Poor) Usage with an Intermediate Temporary: Correct references to the specific instances of the **Record** DSECT can be obtained (apparently) by using an intermediate temporary storage area:

```
         USING Record,7                Map Old instance of Record
         MVC   Temp(L'RecID),RecId     Move from Old to Temp
         USING Record,5                Map New instance of Record
         MVC   RecId,Temp              Move from Temp to New
```

Unfortunately, this version fails because if two registers are based at the same location, the higher-numbered register will be used for calculating displacements. Thus, the second MVC instruction will move the data from Temp back to where it started in the Old record!

The solution for ordinary USINGs is to insert a DROP statement for GR7:

[280] To avoid the danger of improper resolutions using base register zero, a *very* clever programmer observed that setting a large absolute offset in the USING statement and in the manually calculated displacement avoids contaminating later resolutions intended for GR0:

```
USING Record,5           Map New instance of Record
USING 0+X'F999',7        Map Old instance of A (differently)
MVC   RecID,RecID-Record+X'F999'   Move from Old to New
```

The code is correct, but at the cost of complexity and obscure coding unlikely to be understood by later maintainers.

```
         USING Record,7             Map Old instance of Record
         MVC   Temp(L'RecID),RecId  Move from Old to Temp
         Drop  7                    Delete mapping of Old instance
         USING Record,5             Map New instance of Record
         MVC   RecId,Temp           Move from Temp to New
```

In summary, the defects of these two techniques are

- greater likelihood of undetected error;
- deeper understanding required of language details;
- more complex coding;
- less efficient instruction sequences;
- more difficult maintenance.

Example 5 — Correct (But Not Recommended) Usage with Duplicated DSECTs: A programmer observing the defects of the above methods of managing two instances of the **Record** DSECT might decide that the best solution is to make a second copy with a different name, to avoid having to write confusing USING and DROP statements. Thus, he might define an exact copy of **Record** now named **Record2**:

```
Record2  DSect ,                 Record description
RecType2 DS    CL10              Record type
RecID2   DS    CL4               Record ID
RecName2 DS    CL40              Name
RecAddr2 DS    CL66              Address
         - - -                   etc.
RecLen2  Equ   *-Record2         Record length
```

Then, the desired instruction sequence is much cleaner and simpler:

```
         USING Record2,7         Map Old instance of Record (Record2)
         USING Record,5          Map new instance of Record
         MVC   RecID,RecID2      Move from Old to New
```

While it gives the desired sequence, this technique can lead to difficulties in maintenance if the maintainer doesn't appreciate that **Record2** must be an *exact* duplicate of **Record**. If changes are made to the **Record** DSECT, the differences in DSECT and symbol naming make it easy to overlook the need to make equivalent and identical changes to **Record2**.

It is also less obvious that the symbols in this code fragment actually refer to the same objects.

In summary, the defects of this technique are

- greater likelihood of maintenance problems;
- greater difficulty in understanding the code.

Example 5a — Another (But Still Not Recommended) Way with Macro-Duplicated DSECTs: Occasionally, this "duplicate definition" technique is encapsulated in a macro definition. If someone writes a macro named DDSECT to define copies of the **Record** DSECT, the macro can generate as many copies of the DSECT as needed, adding a specified prefix to each of the generated symbols, as illustrated in the following (where the + sign is the Assembler's indication that the statement was generated by a macro-instruction):

```
         DDSECT Prefix=Old        Record DSECT, symbols prefixed 'Old'
+OldRec  DSECT
+OldType DS    - - -
+OldID   DS    CL4
+        - - -
         DDSECT Prefix=New        Record DSECT, symbols prefixed 'New'
+NewRec  DSECT
+NewType DS    - - -
+NewID   DS    CL4
+        - - -
         USING NewRec,5
         USING OldRec,7
         MVC   NewID,OldID        Move RecID from OldRec to NewRec
```

This technique (the most satisfactory of the approaches discussed so far) ensures that only a single source file containing the DSECT's definition is maintained (inside the macro). The defects of this approach are:

- it introduces new symbols and DSECTs into the program, many of which are alternate names for what is really one object;
- it requires that an additional and complex piece of code (the macro definition) be defined and maintained;
- all references to the fields in the DSECT must use prefixed names, even when only a single instance of the object is active (unless a third set of names is generated, with no (or a default) prefix!).

These examples illustrate one of several problems with Ordinary USING statements:

1. You cannot make "simultaneous" symbolic references to a symbol belonging to multiple of instances of a given control section (usually, a DSECT).
2. Ordinary USING statements offer only meager solutions to basic problems of symbolic reference to symbols in multiple instances of a data structure.

We will see other shortcomings that are removed by enhanced USING statements.

Exercises

39.3.1.(1)+ In Figure 629, explain the presence of -4096 and GR8 in the second operand of the MVC instruction.

39.3.2.(2)+ In Figure 627, these statements are said to be a poor programming practice:

```
Using Record,7               Map Old instance of Record
MVC   RecID-Record(5),RecID  Move from Old to New
```

Assuming that RecID-Record=X'00A', what object code is generated by the MVC instruction? Why is it wrong?

39.3.3.(2)+ In Figure 628, the final instruction in Example 2 was written

```
USING Record,7               Map Old instance of Record
MVC   RecID-Record(,5),RecID Move from Old to New
```

Again assuming that RecID-Record=X'00A', what object code is generated by the MVC instruction? Why is it correct?

39.3.4. In Example 2a on page 878 (and in the footnote at the end of the example), what would happen if the LA 1,100 instruction had been written instead as

```
LAY   1,100000          ?
```

39.4. Labeled USING Statements and Qualified Symbols

We want to use fully symbolic references to the fields of the records in Figure 624 on page 876, and not have to use explicit addressing for one of the operands. This capability is provided by Labeled USING statements, which let you symbolically reference more than one instance of a given DSECT at the same time.

Unlike ordinary USING statements, Labeled USING statements have a name field entry. The name field entry of a Labeled USING is called a *qualifier*, as illustrated in Figure 630.

```
qualifier USING base_location,register(s)   Labeled  USING statement
          USING base_location,register(s)   Ordinary USING statement
```

Figure 630. Labeled USING statement syntax

A key concept in using Labeled USING statements is the *qualifier* or *qualifying label*. A qualifier follows the rules for proper forms of ordinary symbols. A symbol is defined to be a qualifier only by its appearance in the name field of a USING statement, and it may not be used as an ordinary symbol. The presence of this name field symbol distinguishes Labeled USINGs from other USING statements.

39.4.1. Qualified Symbols

A *qualified symbol* is a pair of symbols separated by a period: the first symbol is the qualifier and the second is the ordinary symbol. The syntax for qualified symbols is shown in Figure 631.

```
qualified_symbol = qualifier.ordinary_symbol
```

Figure 631. Qualified symbol syntax

Some examples of qualified symbols are:

```
         A.B
      Left.Data
    Record.Field4
```

In these examples, the qualifiers are **A**, **Left**, and **Record**; the ordinary symbols are **B**, **Data**, and **Field4**.

Only symbols may be qualified. Other terms may not be qualified.

We can write USING statements defining these qualifiers as in Figure 632:

```
A       Using  Z,5           Qualifier “A”
Left    Using  Block,9       Qualifier “Left”
Record  Using  Mapping,3     Qualifier “Record1”
```

Figure 632. Examples of qualifier definitions

If a qualifying symbol is *not* present, the USING statement is interpreted by the Assembler as an ordinary USING. Because qualifiers are kept in the same symbol table as ordinary symbols, they must be distinct from other symbols. Thus, a qualified symbol like **X.X** is invalid.

The resolution rule for Labeled USINGs and qualified symbols is simple: if a symbol is qualified, it may be resolved *only* with respect to the base register(s) specified in the Labeled USING statement with the qualifier label.

Symbol qualification is no guarantee of addressability! Address resolution still requires that displacements not exceed 4095 (for unsigned 12-bit displacements) or ±500K (for signed 20-bit displacements), and that the relocation attributes of the addressing expression and the base location in the USING statement must match.

Remember!

Base-displacement resolution of qualified symbol operands is restricted to symbolic operands whose qualifier matches the qualifier defined on a valid Labeled USING statement.

Labeled USINGs provide a clean and simple solution to our problem of copying the **RecID** field from the **Old** record to the **New**. Suppose the **Record** DSECT has been defined as in Figure 624 on page 876. By specifying two Labeled USING statements and by writing qualified symbols, the resulting code is much simpler and easier to understand. Again supposing that GR7 and GR5 contain respectively the addresses of the old and new instances of the record, we can write:

```
Old 1 USING  Record,7            Map old instance of Record
New 2 USING  Record,5            Map new instance of Record
      MVC    New.RecID,Old.RecID  Move field from Old to New
              4        3
```

Figure 633. Copying a field with Labeled USINGs

The Labeled USING with qualifier **Old** (at 1) qualifies the second occurrence of the symbol **RecID** (at 3). Similarly, the Labeled USING with qualifier **New** (at 2) qualifies the first occurrence of the symbol **RecID** (at 4). Because both occurrences of **RecID** are qualified, they can be resolved into base-displacement form *only* with respect to the register with the corresponding qualifier, thus generating the desired `X'D203 500A 700A'`.

Appropriate choices of qualifier names also make the code easier to read and understand!

This example illustrates several advantages of Labeled USINGs:

1. Data objects need be defined only once, no matter how many times they may are referenced concurrently.
2. All references are fully symbolic, and neither explicit base registers nor manually-calculated displacements need to be assigned.
3. The desired, efficient solution is simple, direct, and readable.
4. You need not understand the details of instruction syntax or the address resolution rules for ordinary USING statements.

39.4.2. Dropping a Labeled USING Statement

You can DROP a Labeled USING statement with the DROP statement, but with a qualifier name in place of the register number:

```
         DROP  qualifier
```

Figure 634. DROP statement for Labeled USING

Thus, to DROP the Labeled USINGs in Figure 633, we would write

```
         Drop  Old,New            Drop two Labeled USINGs
```

39.4.3. Labeled USING Statement Summary

Labeled USING statements have interesting properties:

- Though it is written like an ordinary symbol, a qualifier cannot be used both as a qualifier and as an ordinary symbol.
- No symbol without a qualifier matching the qualifying label can be resolved with that USING.
- Because symbol resolution for unqualified symbols and qualified symbols relies on different USING information, you could have more than one USING statement active for a particular base register at the same time. For example:

```
        Using A,9          Ordinary USING
QQ      Using A,9          Labeled  USING, qualifier QQ
        - - -
        LA    0,A+40       Resolved only with Ordinary USING
        LA    1,QQ.A+40    Resolved only with Labeled USING
        Drop  9            Drop ordinary USING; Labeled still active
        LA    2,QQ.A+40    Resolved only with Labeled USING
```

Figure 635. Concurrently active Ordinary and Labeled USINGs

Implied addresses containing symbols without qualifiers will be resolved with the ordinary USING, and qualified symbols will be resolved only with the matching Labeled USING. As Figure 635 shows, the DROP statement deletes the Using Table entry for the ordinary USING, but the Labeled USING remains in effect.

This style of programming should be avoided because it could cause resolution errors as well as making the code more confusing and difficult to understand. Don't use more than a single USING based on a given register.

- DROP statements for Labeled USINGs must be specified by the qualifier, not by the register.
- Normal base-displacement address resolution rules are still in effect:
  - The relocation attribute of the qualified symbol's implied address must match one of the entries in the USING Table before a displacement can be calculated.
  - Valid displacements still cannot exceed 4095 (for 12-bit displacements) or ±500K (for 20-bit displacements).

Warning

An ordinary and a Labeled USING based on the same register should be avoided because it is potentially very confusing and error-prone.

Exercises

39.4.1.(2)+ Suppose you write instructions like these:

```
         Using *,12
         LA    1,ABC
Qual     Using *,9
         LA    2,Qual.ABC
*        - - -
ABC      DS    F
```

What object code will be generated for the two LA instructions?

39.4.2.(2)+ Suppose you write instructions like these:

```
Qual     Using *,9
         Using *,6
         LA    1,ABC
         LA    2,Qual.ABC
*        - - -
ABC      DS    F
```

What object code will be generated for the two LA instructions? How are the Addressing Halfwords resolved, and why?

39.4.3.(2)+ Suppose you used the statements in Figure 635 in a program like this:

```
         Using A,9          Ordinary USING
QQ       Using A,9          Labeled  USING, qualifier QQ
*
         LA    0,A+40       Resolved only with Ordinary USING
         LA    1,QQ.A+40    Resolved only with Labeled USING
         Drop  9            Drop ordinary USING; Labeled still active
         LA    2,QQ.A+40    Resolved only with Labeled USING
*        - - -
A        DS    XL80
```

What addressing halfword will be generated for each of the three LA instructions?

39.4.4.(2) Suppose you wrote

```
A        Equ   1024
Qual     Using 512,7
         LA    3,Qual.A
```

What do you expect the generated object code to be for the LA instruction?

39.5. Dependent USING Statements

Complex data structures often have sub-structures that are described by their own DSECTs. For example, in Figure 624 on page 876, the **RecAddr** field of the **Record** DSECT describes an address. Because an address is itself a complex data structure, it should be defined by its own DSECT, as in Figure 636:

```
Addr     DSect ,                Address-field dummy section
AddrNum  DS    CL8              Street number
AddrStrt DS    CL22             Street name
AddrApt  DS    CL4              Apartment number
AddrCity DS    CL20             City name
AddrStat DS    CL2              State abbreviation
AddrPost DS    CL8              Postal code
PostCtry DS    CL2              Country code
AddrLen  Equ   *-Addr           Length of address field
```

Figure 636. Dummy control section for record address

We should immediately rewrite the **Record** DSECT to utilize the length information in the **Addr** DSECT, as shown in Figure 637:

```
Record   DSect ,                Record description
RecType  DS    CL10             Record type
RecID    DS    CL4              Record ID
RecName  DS    CL40             Name
RecAddr  DS    CL(AddrLen)      Address (length from Addr DSect)
RecPhone DS    CL12             Phone number
         - - -                  etc.
         - - -                  etc.
RecYear  DS    F                Processing year
RecDay   DS    F                Processing day of year
         - - -                  etc.
RecLen   Equ   *-Record         Record length
```

Figure 637. Improved definition of a record description

The only change from Figure 624 is that the length of the **RecAddr** field is now defined by the length of the **Addr** DSECT, rather than by a hand-counted length. This technique should be used whenever possible.

Now, suppose GR7 contains the address of a record in memory, and we must update the postal code field **AddrPost** from the new value stored at **NewPost**. With ordinary USING instructions, we could map the address structure with the **Addr** DSECT, as in Figure 638:

```
Using Record,7          GR7 maps the entire record structure
LA    9,RecAddr         Put address of RecAddr field in GR9
Using Addr,9            Map the address substructure
MVC   AddrPost,NewPost  Update the postal code
Drop  9                 No further updates to Addr fields
```

Figure 638. Mapping a substructure with a second DSECT

But because we have addressability to the entire **Record** structure (provided by GR7), we know the **RecAddr** field is *already* addressable. The displacement of the LA instruction with operand **RecAddr** will be (RecAddr-Record).[281] Unfortunately, we have used a second *and unneeded* base register to map the **Addr** DSECT in the proper position. Assigning an extra register can be quite inconvenient if many of the general registers are already needed for arithmetic or other addresses.

Thus, we want a way to tell the Assembler that it can use a *single* base register (GR7 in Figure 638) to address both the **Record** and **Addr** structures, while letting us make fully symbolic references to fields in the "inner" **Addr** DSECT. Dependent USINGs let us do this.

39.5.1. Definition of Dependent USING Statements

Dependent USING statements look like ordinary USING statements:

```
USING base_location,operand2  Ordinary USING (usually!)
```

but with an important difference!

If the second operand of the USING statement (operand2) is absolute, then it must have a value between zero and fifteen to designate the base register of an ordinary USING statement. However, if the second operand is *relocatable*, it is used as the "base location" at which the first operand is to be "based" or "anchored". The USING statement is then a Dependent USING statement.

This base or anchor location must itself be within the range of, or *depend on* an existing ordinary USING statement: implied operand addresses must still be addressable so they can be resolved into base-displacement form with respect to a declared base register and base location.

The syntax of a Dependent USING statement is therefore:

```
USING base_location,addressable_location  Dependent USING statement
```

Figure 639. Dependent USING statement syntax

Dependent USINGs let you address multiple DSECTs with a single base register.

39.5.2. Examples of Dependent USING Statements

Dependent USING statements allow any object — normally, a DSECT — to be "anchored" or "based" at any location already addressable by an existing USING statement.

We can now revise the instructions in Figure 638 to use a Dependent USING instruction and a single base register, where the two now-unnecessary statements have been "commented out" with the *** asterisks:

---

[281] We have assumed that the **Record** DSECT is less than 4096 bytes long. Even if the **Record** DSECT is much larger, we can either assign more than one base register to provide addressability to all its fields, or use instructions with signed 20-bit displacements.

```
         Using Record,7          GR7 maps the entire record structure
***      LA    9,RecAddr         Put address of RecAddr field in GR9
         Using Addr,RecAddr      Map the address substructure
         MVC   AddrPost,NewPost  Update the postal code
***      Drop  9                 No further updates to Addr fields
```

Figure 640. Anchoring an internal DSECT with a Dependent USING

One fewer instruction and one fewer base register are needed now.

The Assembler requires the second operand of a Dependent USING to be addressable at the time the Dependent USING is encountered. This addressability may actually support more than one Dependent USING. For example, suppose we want to describe two subordinate DSECTs **B** and **C** within an outer DSECT **A**, as sketched in Figure 641:

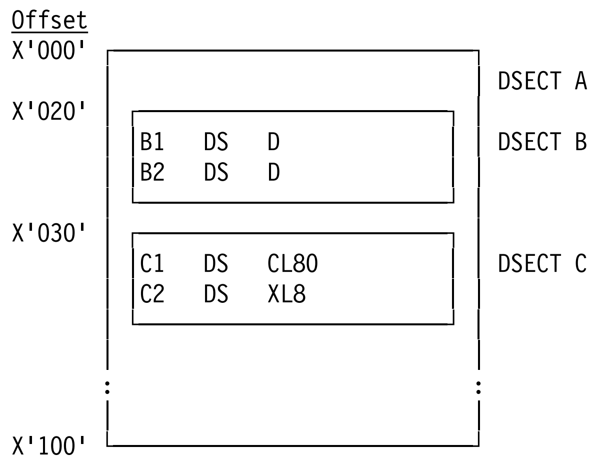


Figure 641. Outer DSECT with two nested DSECTs

The Assembler listing extract in Figure 642 shows how addresses of various fields are resolved:

- The ordinary USING at statement 9 maps the location of the outer DSECT **A**, to provide addressability for all three DSECTs.
- The Dependent USING at statement 10 anchors DSECT **B** at offset X'20' from the start of DSECT **A**.
- DSECT **C** is anchored by the Dependent USING at statement 14 at offset X'10' from the start of DSECT **B**.

```
      R:F  00000           9        USING  A,15       Ordinary: Addr(A) in R15
   1  F 020  00000 00020  10        USING  B,A+32     Dependent: B offset X'20' from A
                          11 *
00058 4100 F028     00008  12        LA     0,B2       B2 at offset X'28' from A
                          13 *
   2  F 030  00000 00010  14        USING  C,B+16     Dependent: C offset X'10' from B
                          15 *
0005C 4100 F080     00050  16        LA     0,C2       C2 at offset X'80' from A
                          17 *
00000                     18 B      DSECT
00000                     19 B1     DS     D          Offset    0  from B
00008                     20 B2     DS     D          Offset    8  from B
                          21 *
00000                     22 C      DSECT
00000                     23 C1     DS     CL80       Offset    0  from C
00050                     24 C2     DS     XL8        Offset X'50' from C
                          25 *
00000                     26 A      DSECT
00000                     27        DS     XL256
```

Figure 642. Assembler listing of multiple Dependent USINGs and DSECTs

In statement 10, the Assembler indicates (at **1**) that the anchoring base location of the first Dependent USING is at offset X'020' from the location specified for base register X'F'. Statement 14 shows (at **2**) that the anchoring base location of the second Dependent USING is at offset X'30'.[282] Even though three distinct DSECTs are referenced, only one base register (GR15) is needed for the two LA instructions.

If the three DSECTs in Figure 641 are mutually independent, they need not be “nested” and can still be defined to use only a single base register, as sketched in Figure 643:

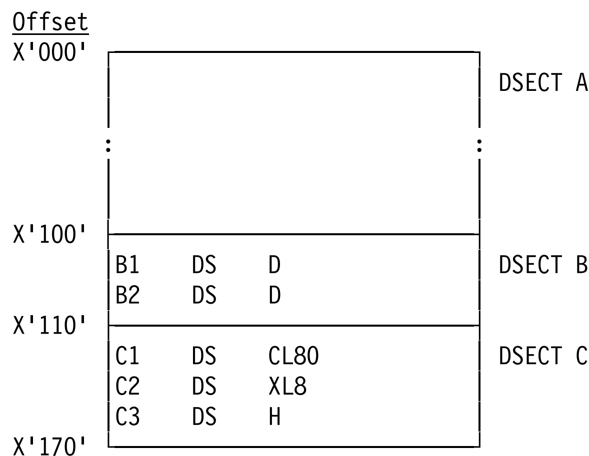


Figure 643. Three independent data structures with one base register

We can define the three independent data structures mapped by DSECTs **A**, **B**, and **C** with these statements:

[282] Unfortunately, two of the USING statements in Figure 642 use absolute offsets. See Exercise 39.5.4 for a better approach.

```
B        DSect
B1       DS    D
B2       DS    D
         DS    0D                    Round up length to a doubleword
BLen     Equ   *-B
C        DSect
C1       DS    CL80
C2       DS    XL8
C3       DS    H
         DS    0D                    Round up length to a doubleword
CLen     Equ   *-C
A        DSect
         DS    XL256
         DS    0D                    Round up length to a doubleword
ALen     Equ   *-A
```

Figure 644. Defining DSECTs for three independent data structures

Unlike the DSECT definitions in Figure 642, each DSECT defines a symbol for its length after rounding to a doubleword boundary. This is a good practice in case you need to allocate storage for multiple structures to be based on a single base register, because you should ensure proper alignment of each structure. Suppose we allocate enough storage for all three structures:

```
TotalLen Equ   ALen+BLen+CLen       Total amount of storage needed
```

Now, suppose GR7 contains the address of **TotalLen** allocated bytes aligned on a doubleword boundary. We can use one ordinary and two Dependent USINGs to map all three structures:

```
*        'TotalLen' bytes addressed by GR7
         - - -
         Using A,7                   Map DSECT A first
         Using B,A+ALen              Map DSECT B following A
         Using C,B+BLen              Map DSECT C following B
         - - -
```

Figure 645. Defining a mapping of three independent but contiguous data structures

Instructions referencing any of the fields within the three structures will be resolved with base register 7, assuming that **TotalLen** is small enough that a single base register can address all the fields.

39.5.3. Mapping a CSECT as a DSECT

Dependent USINGs let you do very interesting things. For example, suppose you need to construct a message into which several insertions will be made in known positions. First, construct the message skeletons in a separate CSECT in the same assembly as the **Program** CSECT:

```
MsgSkels CSect ,                         Message skeletons
Msg1     DC    C'This message for '      Message 1
Msg1To   DC    C'totototo'               Modified field: 'To' name
         DC    C' is from '              Additional message characters
Msg1From DC    C'fromfrom'               Modified field: 'From' name
Msg1L    Equ   *-Msg1                    Length of message 1
         - - -
         - - -                           Other messages follow
```

Figure 646. Example of a message-skeleton CSECT

We complete the message in the **Program** CSECT by first moving the message skeleton from the **MsgSkels** CSECT to an output area named **OutMsg**. Then, the Dependent USING maps the structure of the skeleton on the **OutMsg** where the completed message is constructed:

```
Program  CSect ,
         BASR  12,0                  Set base register
         Using *,12                  Provide local addressability
         L     10,AddrMsgs           Point to messages
         L     11,AOutMsg            Point to buffer area
         Using MsgSkels,10           Addressability for Messages CSECT
         Using OutMsg,11             Addressability for OutMsg area
         MVC   OutMsg(Msg1L),Msg1    Move skeleton to buffer
         Drop  10                    Don't reference original skeleton
         Using Msg1,OutMsg           Map original skeleton onto buffer
         MVC   Msg1To,ToName         Move "To" name
         MVC   Msg1From,FromName     Move "From" name
         Drop  11                    Done with buffer addressing
         - - -
ToName   DC    CL(L'Msg1To)'You'     Addressee name
FromName DC    CL(L'Msg1From)'Me'    Sender name
AddrMsgs DC    A(MsgSkels)           Address of Messages CSECT
AOutMsg  DC    A(OutMsg)             Address of OutMsg area
         - - -
OutMsg   DS    CL121                 Area for constructing messages
```

Figure 647. Example of mapping a CSECT as though it is a DSECT

The key statement is the Dependent USING statement:

```
         Using Msg1,OutMsg           Map original skeleton onto buffer
```

directs the Assembler to resolve the first-operand addresses in the two following MVC instructions with offsets defined in the **MsgSkels** CSECT, but mapped on the **OutMsg** area. Note that when those two MVCs are executed, the **MsgSkels** CSECT is no longer addressable! The Dependent USING statement mapping the skeleton constant onto the **OutMsg** buffer avoids the need to define a DSECT for each message, and also avoids manually calculating displacements and assigning base registers.

39.5.4. Dropping Dependent USINGs

Unlike ordinary and Labeled USING statements, you must DROP a Dependent USING by dropping the ordinary USING on which it depends. For example, in Figure 641 on page 887, we can't remove addressability to DSECT **C** separately; we must DROP the USING statement that provides addressability for all three DSECTs. In Figure 647, to drop the USING in the statement

```
         Using Msg1,OutMsg       Map original skeleton onto buffer
```

we must drop the USING on which it depends:

```
         Drop  11                Drop GR11
```

because the mapping of **Msg1** is anchored at **OutMsg** which in turn is based on GR11.

We'll see in Section 39.6 how Labeled Dependent USING statements can help.

39.5.5. Dependent USING Statement Summary

Dependent USINGs can provide elegant solutions to problems involving the management of data structures that are adjacent, nested, or overlapping in storage.

- They provide addressability with a minimum number of registers.
- They allow fully symbolic structure and substructure mappings with independent DSECTs.
- They encourage simple mappings of complex data structures.
- They let you map “variant” records in which the structure of parts of the record depend on preceding data values, with different mappings along different code paths.

One way to view the difference between ordinary Dependent USINGs is by the way the first-operand location is “based” or “anchored”.

- For ordinary USINGs, the first operand is “based” on the *base register* specified by the second operand. Instructions set the register contents at *execution* time.
- For dependent USINGs, the first operand is “based” or “anchored” on the *location* specified by the second operand; this location must already be addressable. The relative position of the first operand is set at *assembly* time by the Assembler.

Exercises

39.5.1.(2)+ Using the instructions sketched in Figures 646 and 647, create a program that can be correctly assembled. Study the displacements in the MVC instructions to verify that the intended data is moved to the correct fields.

39.5.2.(2) In Figure 643, will it make any difference if the three DSECTs are arranged in a different order?

39.5.3.(2)+ The first paragraph in Section 39.5.3 says that the `MsgSkels` CSECT containing the message skeletons is in the same assembly as the `Program` CSECT. What will happen if it is in a separate assembly?

39.5.4.(2)+ Revise the DSECT statements in Figure 642 on page 888 to make all references fully symbolic, without offsets like `A+32` and `B+16`.

39.6. Labeled Dependent USING Statements

Labeled Dependent USINGs combine the benefits of Labeled USINGs and Dependent USINGs:

- multiple copies of an object may be active simultaneously (Labeled)
- many objects may be addressed with a single base register (Dependent).

We start with several examples that illustrate typical problems when we use ordinary USINGs, and how easily we can solve them with Labeled Dependent USINGs.

The syntax of a Labeled Dependent USING requires a qualifying label in the name field of the USING statement, and a relocatable second operand indicates where the first operand is to be “anchored” or “based”.

```
qualifier USING base_location,addressable_location  Labeled Dependent USING
```

Figure 648. Labeled Dependent USING statement syntax

As with unlabeled Dependent USINGs, the second operand must be addressable with reference to an ordinary USING statement somewhere earlier in the program.

Suppose we have a data structure containing several data items, including two identical substructures described by a single DSECT. The substructure is defined by a DSECT named **Inner**:

```
Inner    DSect ,
IVarJ    DS    XL5
IVarK    DS    XL7
InnerLen Equ   *-Inner             Length of Inner structure
```

and the containing data structure is described by a DSECT named **Outer**:

```
Outer    DSect ,
OutVarA  DS    CL12
Out_Inr1 DS    XL(InnerLen)        First substructure
OutVarB  DS    CL33
Out_Inr2 DS    XL(InnerLen)        Second substructure
OutVarC  DS    XL3
OuterLen Equ   *-Outer             Length of Outer structure
```

Figure 649. Nesting two identical structures within a third

Out_Inr1 and **Out_Inr2** are the positions within the **Outer** DSECT where the two sub-structures are mapped.

We will examine three approaches to managing the description and addressing of the data elements in these structures:

- first, we consider ordinary USINGs and the problems they present;
- second, we examine Labeled USINGs;
- third, we will see how Labeled Dependent USINGs provide a solution free of the defects of the two previous approaches.

39.6.1. Nesting Structures Addressed with Ordinary USINGs

To address the three structures with ordinary USINGs, we need to provide three base registers and three USING statements. As in Section 39.4, we want to manage two active instances of the **Inner** DSECT, but only one active instance is allowed with ordinary USINGs.

We could make a second copy of the **Inner** DSECT, and then address it and the “original” copy with separate USINGs.

```
Znner    DSect ,
ZVarJ    DS    XL5
ZVarK    DS    XL7
ZnnerLen Equ   *-Znner             Length of Znner structure
```

The three DSECTs can now be addressed with statements like the following:

```
         Using Outer,10            GR10 points to the Outer DSECT
         LA    11,Out_Inr1         GR11 points to 1st copy of Inner
         Using Inner,11            USING for 1st copy of Inner
         LA    12,Out_Inr2         GR12 points to 2nd copy of Inner
         Using Znner,12            USING for 2nd copy (now named Znner)
```

We've already described the defects and difficulties with ordinary USINGs in this context: maintenance and readability problems are much greater when more than one name is used for the same thing.

39.6.2. Nesting Structures Addressed with Labeled USINGs

A somewhat better solution involves Labeled USINGs. They allow the two instances of the **Inner** DSECT to be addressed using a single definition of **Inner**, eliminating any need for a **Znner** DSECT:

```
         Using Outer,10            GR10 points to Outer DSECT
         LA    11,Out_Inr1         GR11 points to 1st copy of Inner
In1      Using Inner,11            Labeled USING for 1st copy of Inner
         LA    12,Out_Inr2         GR12 points to 2nd copy of Inner
In2      Using Inner,12            Labeled USING for 2nd copy of Inner
```

The implied addresses in both LA instructions will be resolved with register 10 as the base register.

The remaining defect in this example is the need to use three addressing registers when only one is actually needed. Labeled Dependent USINGs provide the desired saving.

39.6.3. Nested Structures Addressed with Labeled Dependent USINGs

The best solution involves Labeled Dependent USINGs: they allow the entire structure and all its components to be addressed with the minimum number of registers, and with proper naming for all components.

Assume again that the address of the containing **Outer** structure is placed in GR10; then, we can write:

```
         Using Outer,10          GR10 addresses Outer DSECT
In1      Using Inner,Out_Inr1    Labeled Dependent USING for 1st Inner
In2      Using Inner,Out_Inr2    Labeled Dependent USING for 2nd Inner
```

Figure 650. Addressing two nested DSECTs with Labeled Dependent USINGs

The first instance of **Inner** is "anchored" at **Out_Inr1**, and references to its components are made using qualifier **In1**. Similarly, the second instance of **Inner** is anchored at at **Out_Inr2**, and its components can be qualified with **In2**. References to the fields in the three structures can then be made freely, and only a single register is needed to address the entire structure. Figure 651 shows how references can be made to fields within the three DSECTs:

```
         CLC   In1.IVarK,In2.IvarK    Compare the two IVarK fields
         - - -
         MVC   OutVarC,In2.IVarK      Copy 3 bytes of 2nd IVarK
         - - -
         PACK  In1.IvarK,OutVarC      Pack OutVarC to 1st IVarK
```

Figure 651. Data in nested DSECTs addressed with Labeled Dependent USINGs

39.6.4. Multiple Nesting of Identical Structures

As the number and nesting of data structures increases, addressing the components becomes more difficult. For example, suppose we wish to create a data structure in which an outermost structure described by the **Top** DSECT contains three copies of a structure described by the **Mid** DSECT, and each of those contains three copies of a structure described by the **Bot** DSECT. The complete structure might look somewhat like Figure 652, where the three **Mid** DSECTs are identified as **M1**, **M2**, and **M3**, and the **Bot** DSECTs are identified as **B1**, **B2**, and **B3**.

Top	Mid M1	Bot B1
		Bot B2
		Bot B3
	Mid M2	Bot B1
		Bot B2
		Bot B3
	Mid M3	Bot B1
		Bot B2
		Bot B3

Figure 652. Multiply-Nested Data Structures

If ordinary USINGs address all components of this set of structures, we will need *thirteen* base registers! This is beyond the capabilities of most programs, so that we would be forced to use "unnatural" solutions if we are restricted to ordinary USING statements.

Dependent USINGs will help only a little, because of the high degree of repetition among the inner structures.[283]

[283] This example is rather artificial, but helps show the power of Labeled Dependent USINGs. Two more-realistic examples are described in Sections 39.6.6 and 39.7.

Suppose the three DSECTs named **Top**, **Mid**, and **Bot** in Figure 652 are described by the DSECTs in Figure 642:

```
Bot      DSect ,               Third-level DSECT (bottom level)
X1       DS    XL5               First data element
X2       DS    XL5               Second data element
         DS    0D              Round length to a doubleword
L_Bot    Equ   *-Bot           Length of Bot DSect

Mid      DSect ,               Second-level DSECT (middle level)
MidVar1  DS    CL40              Data in second-level DSect
B1       DS    XL(L_Bot)         First third-level DSect
MidVar2  DS    CL60              Data in second-level DSect
B2       DS    XL(L_Bot)         Second third-level DSect
MidVar3  DS    CL20              Data in second-level DSect
B3       DS    XL(L_Bot)         Third third-level DSect
MidVar4  DS    CL30              Data in second-level DSect
         DS    0D              Round length to a doubleword
L_Mid    Equ   *-Mid           Length of Mid DSect

Top      DSect ,               First-level DSECT (top level)
M1       DS    XL(L_Mid)         First second-level DSect
M2       DS    XL(L_Mid)         Second second-level DSect
M3       DS    XL(L_Mid)         Third second-level DSect
         DS    0D              Round length to a doubleword
L_Top    Equ   *-Top           Length of Top DSect
```

Figure 653. Doubly Nested DSECT definitions

Addressing this structure with ordinary USINGs is nearly impossible to do cleanly, and a solution with Labeled USINGs also requires *thirteen* registers to address the thirteen different active DSECTs.

Labeled Dependent USINGs provide the only manageable solution. Though the example in Figure 654 looks somewhat complicated, it has a simple basic structure.

```
         Using Top,7                  1 Top level
*                                     |
Mid1     Using Mid,M1       1         |  2 Map Mid into Top at M1
*                                     |  |  Bottom level:
M1B1     Using Bot,Mid1.B1    2       |  |  3 Map Bot into Mid1 at B1
M1B2     Using Bot,Mid1.B2    2       |  |  3 Map Bot into Mid1 at B2
M1B3     Using Bot,Mid1.B3    2       |  |  3 Map Bot into Mid1 at B3
*                                     |  2 Middle level:
Mid2     Using Mid,M2       1         |  | Map Mid into Top at M2
*                                     |  |  Bottom level:
M2B1     Using Bot,Mid2.B1    3       |  |  3 Map Bot into Mid2 at B1
M2B2     Using Bot,Mid2.B2    3       |  |  3 Map Bot into Mid2 at B2
M2B3     Using Bot,Mid2.B3    3       |  |  3 Map Bot into Mid2 at B3
*                                     |  2 Middle level:
Mid3     Using Mid,M3       1         |  | Map Mid into Top at M3
*                                     |  |  Bottom level:
M3B1     Using Bot,Mid3.B1    4       |  |  3 Map Bot into Mid3 at B1
M3B2     Using Bot,Mid3.B2    4       |  |  3 Map Bot into Mid3 at B2
M3B3     Using Bot,Mid3.B3    4       |  |  3 Map Bot into Mid3 at B3
```

Figure 654. Addressing doubly nested DSECT definitions

The three Labeled Dependent USING statements in Figure 654 (tagged **1**) map the middle-level DSECT **Mid** into the outermost DSECT **Top**. Because there will be three instances of **Mid** simultaneously active, the qualifiers **Mid1**, **Mid2**, and **Mid3** distinguish the first, second, and third

instances of **Mid** within **Top**. The three instances of **Mid** are anchored at the positions within **Top** defined by the fields named **M1**, **M2**, and **M3** respectively.

The three innermost instances of the **Bot** DSECT are mapped into the three instances of **Mid** similarly. For example, the three Labeled Dependent USING statements for the first instance of **Mid** (tagged **2**) anchor the three instances of **Bot** within **Mid** at the positions named **B1**, **B2**, and **B3** respectively. (Referring to Figure 652 may help.)

Because there will be three different active instances of those labels, we use the qualifier **Mid1** to qualify the references to **B1**, **B2**, and **B3**. Thus the second operand of each of the three Labeled Dependent USING statements tagged **2** is qualified with **Mid1**. The qualifiers on those three USINGs, **M1B1**, **M1B2**, and **M1B3**, are used to qualify references to the first three of the nine possible instances of the fields **X1** and **X2**. The qualifier **M1B3** means "first **Mid** and third **Bot**". This shows how appropriately chosen qualifiers can help you to understand complex structures more easily.

The mappings of the second and third sets of instances of the **Bot** DSECT are defined similarly, in the sets of three Labeled Dependent USINGs tagged **3** and **4**. The qualifiers **M2B1** through **M3B3** are then used to qualify references to the fields within the DSECT named **Bot**.

We can then write instructions to reference these fields, with appropriate qualifiers, as illustrated in Figure 655. All symbolic references will be resolved with the base register (or registers) specified in the USING statement for **Top**.

```
*        Move fields 'X1' and 'X2' within 'Bot' DSECTs
         MVC   M1B1.X1,M1B1.X2         Within bottom-level DSECT M1B1
         MVC   M1B3.X2,M1B1.X1         Across bottom-level DSECTs in M1
         MVC   M3B2.X2,M3B3.X2         Across bottom-level DSECTs in M3
         MVC   M2B1.X1,M3B2.X2         Across bottom-level DSECTs in M2, M3

*        Move complete 'Bot' DSECTs within 'Mid' DSECTs
         MVC   Mid3.B1,Mid3.B3         Within mid-level DSECT Mid3
         MVC   Mid1.B3,Mid2.B1         Across mid-level DSECTs Mid1, Mid2

*        Copy fields across 'Mid' DSECTs
         MVC   Mid2.MidVar1,Mid3.MidVar1    Across two middle DSECTs

*        Move complete 'Mid' DSECTs within 'Top' DSECT
         MVC   M1,M2                   Across top-level DSECTs Top
```

Figure 655. Using the Labeled Dependent USINGs to move data

As you can appreciate, coding instructions like these with ordinary USING statements would be much more difficult to write and understand.

Remember!

Qualified symbols may be used to declare the "anchor" operand in a Labeled Dependent USING statement that itself defines another qualifier!

39.6.5. Mapping an Array of Identical Data Structures

Suppose you have a small array of identical data structures, and you want to refer to fields within different array elements. First, you could define a DSECT describing the data structure:

```
Struc    DSect ,                 Structure of an array element
StrF1    DS    CL8               First field
StrF2    DS    F                 Second field
StrF3    DS    A                 Third field
LStruc   Equ   *-Struc           Structure Length
```

Then, suppose GR9 contains the address of the first element of the array. You can then map the first few elements of the array with Labeled Dependent USINGs:

```
EL1      Using Struc,9                    Map first  element
EL2      Using Struc,EL1.Struc+1*LStruc   Map second element
EL3      Using Struc,EL1.Struc+2*LStruc   Map third  element
EL4      Using Struc,EL1.Struc+3*LStruc   Map fourth element
EL5      Using Struc,EL1.Struc+4*LStruc   Map fifth  element
         - - -                            etc.
```

Then, you can refer to fields within each element of the array:

```
         L     1,EL3.StrF2          Get field 2 from element 3
         A     1,EL5.StrF3          Add field 3 from element 5
         MVC   EL2.StrF1,EL4.StrF1  Move field 1 from element 4 to 2
```

This technique is limited to small arrays because the number of Labeled Dependent USING statements grows with the number of elements to be referenced. If you need to refer to fields in two widely separated elements, it may be easier to assign a base register to each and map the elements with Labeled USINGs instead.

39.6.6. Two MVS Data Control Blocks (DCBs) in a Program

This small example shows how you can benefit from Labeled Dependent USINGs in a program, where the program base register also addresses two embedded DSECT-mapped structures.

Many programs contain two or more Data Control Blocks (DCBs) for input and output data sets, and are often coded in the same program as the instructions that refer to them. If only ordinary USINGs are available, (at least) three registers must be used for addressability: one (or more) for the program itself, and one for each DCB. Furthermore, only one of the DCBs can be mapped with the IBM-provided IHADCB DSECT[284] because both cannot be mapped simultaneously with distinct registers.

With ordinary USING statements, a typical instruction sequence to copy the two-byte logical record length (DCBLRECL) from the input DCB to the output DCB might look like this:

```
         Using *,12               Program base register
         - - -
         LA    3,OutDCB           Point GR3 to Output DCB
         LA    2,InDCB            Point GR2 to Input DCB
         Using IHADCB,2           Map Input DCB
         MVC   DCBLRECL-IHADCB(,3),DCBLRECL  Copy InDCB LRECL to OutDCB
         - - -
InDCB    DCB   DDNAME=..., etc.   Input DCB
OutDCB   DCB   DDNAME=..., etc.   Output DCB
         - - -
         DCBD  ...,etc.           Generate IHADCB DSECT
```

Figure 656. Addressing two DCBs with ordinary USINGs

Three registers must be assigned, and one of the operands in the MVC instruction must be written with explicitly assigned base and displacement.

It's better to make symbolic references to fields in both DCBs at the same time. Iin Figure 657, the two Labeled Dependent USINGs with qualifiers **In** and **Out** let you make fully symbolic references to both DCBs, and without needing additional base registers.

[284] Generated by the DCBD macro. (IHADCB is sometimes nicknamed “I Had A Control Block”.)

```
         Using *,12              Program base register
         - - -
In    1  Using IHADCB,InDCB      Labeled Dependent USING
Out   2  Using IHADCB,OutDCB     Labeled Dependent USING
         - - -
         MVC   Out.DCBLRECL,In.DCBLRECL  Addresses resolved with GR12
                2            1
         - - -
InDCB    DCB   DDNAME=..., etc.  Input DCB
OutDCB   DCB   DDNAME=..., etc.  Output DCB
         - - -
         DCBD  ...,etc.          Generate IHADCB DSECT
```

Figure 657. Addressing instructions and DCBs with one register

The base address in the first USING can be "anchored" at any addressable location that provides addressability to the two DCBs and the MVC instruction.

To DROP a Labeled Dependent USING, simply DROP the qualifier, as is done for a Labeled USING.

Exercises

39.6.1.(2) Referring to Figure 652, sketch the movement of data caused by the instructions in Figure 655.

39.6.2.(1) In Figure 649, write ordinary USING statements to provide addressability to all three DSECTs.

39.6.3.(2)+ In Figures 653 and 654, what are the offsets from the origin of the **Top** DSECT of these qualified symbols?

- M1B1.X1
- Mid1.MidVar1
- M3B2.X2
- Mid3.MidVar4

39.6.4.(3)+ Create a complete program and assemble the statements in Figures 653, 654, and 655. Study the generated object code to verify that all addressing halfwords are generated correctly.

39.7. Example of a Large "Personnel-File" Record (*)

The power of Dependent and Labeled Dependent USINGs is most evident when you must handle complex data records, especially when the structure of fields in later parts of the record depends on data values in earlier fields, or where repeated identically-structured fields (mapped by the same DSECT) appear several times within the record's structure.

Note: This example, while rather complex, shows how you can use the Assembler's Dummy Control Sections and Labeled Dependent USINGs for tasks once thought to be manageable only by a high-level language.

Suppose our program must refer to various fields in records maintained in a personnel file. Each record contains information about an employee, and fields within the record contain different kinds of information.

First, we define the layout of the employee record with an **Employee** DSECT. All symbols start with the letter E.

```
Employee DSect ,                  Employee record
EPerson  DS    CL(PersonL)        Person field
EHire    DS    CL(DateL)          Date of hire
EWAddr   DS    CL(AddrL)          Work (external) address
EPhoneW  DS    CL(PhoneL)         Work telephone
EPhoneF  DS    CL(PhoneL)         Work Fax telephone
EMarital DS    X                  Marital Status
ESpouse  DS    CL(PersonL)        Spouse field
E#Deps   DS    PL2                Number of dependents
EDep1    DS    CL(PersonL)        Dependent 1
EDep2    DS    CL(PersonL)        Dependent 2
EDep3    DS    CL(PersonL)        Dependent 3
EmployeL EQU   *-Employee         Length of Employee record
```

Figure 658. Define a personnel-file record

The record contains information about the employee: a description of the person, the employee's spouse and first three dependents, work address, date of hire, work telephone, and so forth. Space is provided in the **Employee** DSECT for several other “nested” or “overlaid” DSECTs described below.

The description of each person (employee, spouse, dependents) is similarly defined by a **Person** DSECT shown in Figure 659:

```
Person   DSect ,                  Define a "Person" field
PFName   DS    CL20               Last (Family) name
PGName   DS    CL15               First (Given) name
PInits   DS    CL3                Initials
PDoB     DS    CL(DateL)          Date of birth
PAddr    DS    CL(AddrL)          Home address
PPhone   DS    CL(PhoneL)         Home telephone number
PSSN     DS    CL9                Social Security Number
PSex     DS    CL1                Gender
PersonL  EQU   *-Person           Length of Person field
```

Figure 659. Employee-record Person DSECT

The fields in the **Person** DSECT describe their name, date of birth, home address and telephone, and other items. Again, space has been reserved for three other “nested” DSECTs describing a date, an address, and a telephone number.

Three more DSECTs might be defined as follows. First, the **Date** DSECT:

```
Date     DSect ,                  Define a calendar date field
Year     DS    CL4                YYYY
Month    DS    CL2                MM
Day      DS    CL2                DD
DateL    EQU   *-Date             Length of Date field
         ORG   Date
DateF    DS   0CL(DateL)          Full YYYYMMDD date field
         ORG   ,
```

Figure 660. Employee-record Date DSECT

The last three statements are used to define the symbol **DateF** as a single field containing the entire contents of the three **Date** fields.

The **Addr** DSECT, describing a postal address, is defined in a similar way:

```
Addr     DSect ,                Define an address field
AStr#    DS    CL30             Street number
APOBApDp DS    CL15             P.O.Box, Apartment, or Department
ACity    DS    CL24             City name
AState   DS    CL2              State abbreviation
AZip     DS   0CL9              U.S. Post Office Zip Code
AZipP    DS    CL5              Primary Zip Code
AZipX    DS    CL4              Secondary Zip Code extension
AddrL    EQU   *-Addr           Length of Address field
         ORG   Addr
AddrF    DS   0CL(AddrL)        Full address structure
         ORG   ,
```

Figure 661. Employee-record Address DSECT

Again, the last three statements are used to define the symbol **AddrF** as a single field containing the entire contents of all the **Addr** fields.

Finally, we define the **Phone** DSECT, describing a (U.S.) commercial telephone number:

```
Phone    DSect ,                Define a Telephone field
PhCtry   DS    CL2              Country code
PhArea   DS    CL3              Area Code
PhLocal  DS    CL7              Local number
PhExt    DS    CL4              Extension
PhoneL   EQU   *-Phone          Length of Phone field
         ORG   Phone
PhoneF   DS   0CL(PhoneL)       Full telephone number structure
         ORG   ,
```

Figure 662. Employee-record Phone DSECT

As before, the last three statements define the symbol **PhoneF** to name a single field containing the entire contents of all the **Phone** fields.

At this point, it's worth sketching the nesting of these various DSECTs.

- The **Date** DSECT appears at two different levels of nesting: the Date-of-Hire field (**EHire**) in the **Employee** DSECT is nested two levels deep, and the Date-of-Birth fields (**PDoB**) in each **Person** DSECT are nested three levels deep (because the **Person** DSECT is nested two levels deep in the **Employee** DSECT).
- The **Addr** DSECT is nested two levels deep (as the employee's work address), and three levels deep (as the home-address field (**PAddr**) within each **Person** DSECT).
- Finally, the **Phone** DSECT is also nested two levels deep (**PPhone**, for the employee's home) and three levels deep (**EPhoneW**, the employee's work number).

These nesting levels are shown in the upper right corners of the boxes in Figure 663.

While this example may seem a bit complex, we will use it again in discussing Labeled Dependent USINGs, where the full power of those statements can be shown.

Assume that the **Employee, Person, Date, Addr**, and **Phone** structures have been defined as illustrated in Figures 658 through 662.

First, we show how Dependent USINGs can help with mapping this structure. Suppose such an employee record has been placed in memory and its address is in GR10; we now wish to manipulate various fields within the record. The necessary DSECT addressing can be established as follows:

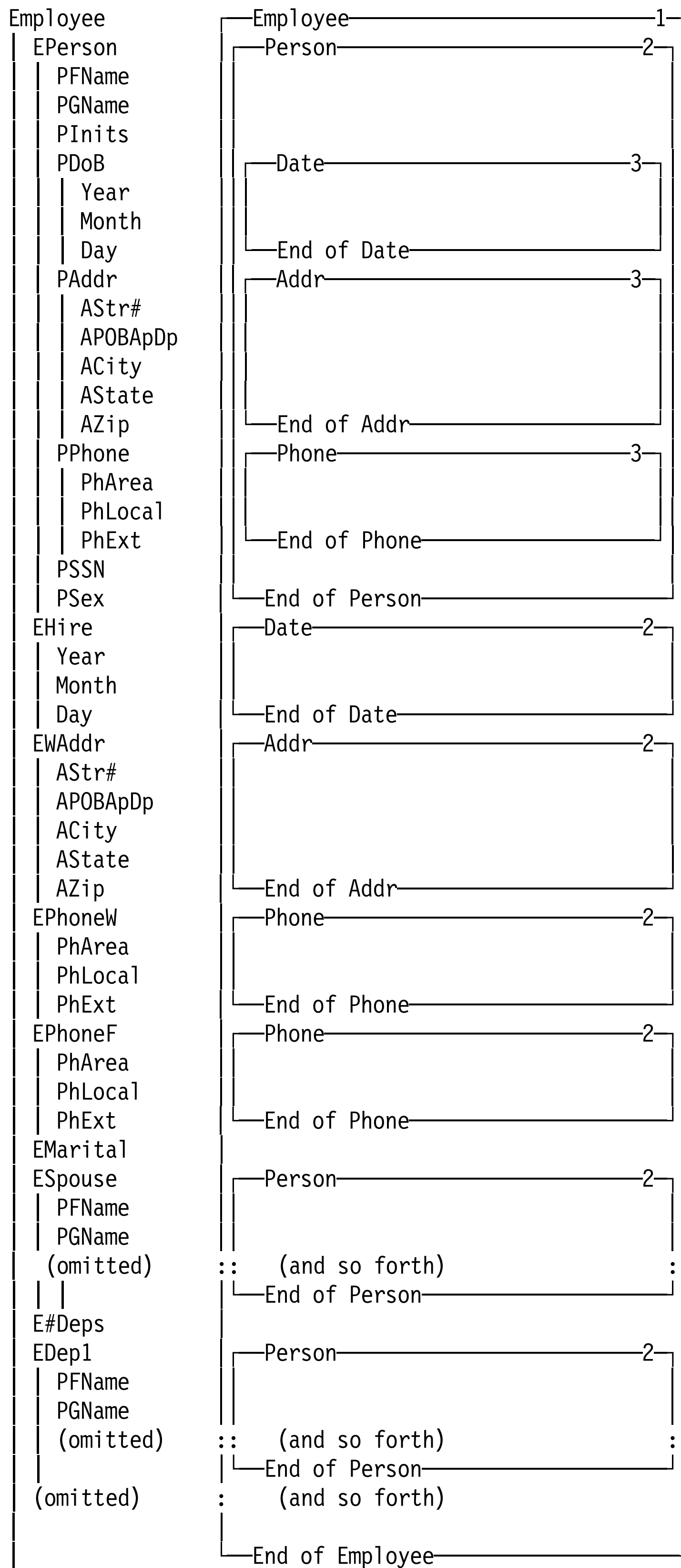


Figure 663. DSECT nesting in an employee record

```
USING Employee,10         R10 points to Employee record
USING Person,EPerson      Anchor Person DSECT at EPerson
USING Date,PDoB           Anchor Date DSECT at PDoB field
USING Addr,PAddr          Anchor Addr DSECT at PAddr field
USING Phone,PPhone        Anchor Phone DSECT at PPhone field
```

Figure 664. Anchoring various DSECTs within Employee record

The five USING statements in Figure 664 provide addressability to five different DSECTs:

- The **Employee** DSECT is based on an ordinary USING statement with base register GR10. All other implied address resolutions within the **Employee** DSECT will be resolved using GR10 as the base register.
- The **Person** DSECT is anchored by the first Dependent USING at the Eperson field in the **Employee** DSECT.
- Within the **Person** DSECT, the **Date** DSECT is anchored by the second Dependent USING at the **PDoB** field in the **Person** DSECT.
- Within the **Person** DSECT, the **Addr** DSECT is anchored by the third Dependent USING at the **PAddr** field in the **Person** DSECT.
- Within the **Person** DSECT, the **Phone** DSECT is anchored by the fourth and last Dependent USING at the **PPhone** field in the **Person** DSECT.

All symbolic references to fields in any of the five DSECTs will be resolved with a single base register, so long as the size of the **Employee** record does not exceed 4096 bytes. If it does, the problem is easy to fix: simply add another base register to the ordinary USING statement in Figure 664, and another 4096 bytes will be addressable automatically.

We can now use these definitions to access and manipulate some fields described by those five DSECTs, as in Figure 665:

```
CLC   PGname,Input_Name       Compare record name to input
- - -
MVC   PhExt,=CL(L'PhExt)' '   Blank phone extension field
- - -
CLC   AZipP,=C'95141'         Check for a specified Postal Zip Code
```

Figure 665. Manipulating fields within an Employee record

The primary limitation of the Dependent USINGs shown in this example is that only a single instance of each DSECT is addressable at any one time. In many applications this may be adequate; if not, Labeled Dependent USINGs will help.

If ordinary USING statements had been required in Figure 664, the resulting burden on the general registers could have been severe. Statements such as these might have been required:

```
USING Employee,10        R10 points to Employee record
LA    9,Eperson          Address of EPerson field
USING Person,9           Anchor Person DSECT at EPerson field
LA    8,PDoB             Address of PDoB field
USING Date,8             Anchor Date DSECT at PDoB field
LA    7,PAddr            Address of PAddr field
USING Addr,7             Anchor Addr DSECT at PAddr field
LA    6,PPhone           Address of PPhone field
USING Phone,6            Anchor Phone DSECT at PPhone field
```

Figure 666. Addressing DSECTs within Employee record with ordinary USINGs

The coding is likely to be less efficient, and the number of opportunities for misunderstanding and error has also increased.

Now, we will consider addressing multiple active instances of the inner structures in this **Employee** record. We assume that the program has placed the **Employee** record into memory at an address referenced by GR10 again.

39.7.1. Personnel-File Record Example: Comparing Birth Dates

Suppose we must compare the birth dates of the employee and the employee's spouse. Because the birth date is a component of the **Person** DSECT, we must establish mappings of the two instances of that DSECT. In Figure 667 this is done with two Labeled Dependent USING statements:

```
* Example 1: Compare Employee and Spouse Family Dates of Birth

         USING Employee,10         Assume R10 points to the record

PE   1   USING Person,EPerson      Overlay Person DSECT on Empl. field
PS   2   USING Person,ESpouse      Overlay Person DSECT on Spouse field

         CLC   PE.PDoB,PS.PDoB     Compare Employee/Spouse birth dates
               1       2
```

Figure 667. Comparing dates of birth in Employee record

The first Labeled Dependent USING statement (**1**) maps the **Person** structure onto the **Employee** record at the position defined by **EPerson** with qualifier **PE**; this describes information about the employee. The second Labeled Dependent USING statement (**2**) maps the **Person** structure onto the **Employee** record at the position defined by **ESpouse** with qualifier **PS**, describing information about the employee's spouse.

The CLC instruction compares the dates. Both instances of the **Person** structure are nested at the same level within the **Employee** structure, so that similar styles of qualification are used for the two occurrences of the symbol **PDoB**.

39.7.2. Personnel-File Record Example: Comparing Different Dates

Suppose our second requirement is to check the employee record to see if the date of birth of the first dependent is later than the employee's date of hire. In this case, we must deal with two different levels of nesting of the **Date** structure: one (the employee's date of hire) is nested directly within the **Employee** DSECT at the position named **EHire**, while the birth date of the first dependent is nested in the **Employee** DSECT at position **EDep1** within the first-dependent **Person** DSECT at position **PDoB**. Thus, we will need additional Labeled Dependent USINGs to properly establish addressability to the PDoB field.

```
* Example 2: Compare Date of Hire to Birth Date of Dependent 1

EHD  3   USING Date,EHire        Overlay Date DSECT on Date of Hire

PD1  4   USING Person,EDep1      Overlay Person DSECT on Dependent 1
DD1  5   USING Date,PD1.PDoB     Overlay Date DSECT on Dependent 1
                    4

         CLC   EHD.DateF,DD1.DateF     Compare hire date to Dep 1 DoB
               3         5

         DROP  EHD,DD1           Remove both date associations
```

Figure 668. Comparing date fields in different parts of an Employee record

In order to map the two instances of the **Date** DSECT, we first issue a Labeled Dependent using with qualifier **EHD** to describe the employee's date of hire (at **EHire**, **3**). Then, to map the first dependent's date of birth, we must first map a **Person** DSECT onto the employee record (at **EDep1**, **4**) with qualifier **PD1**. Finally, within that **Person** DSECT, we describe the dependent's date of birth by mapping the **Date** DSECT onto the **Person** structure (at **PDoB**, **5**) with qualifier **DD1**.

The CLC instruction refers to two complete date fields **DateF**, qualified to associate one with the date of hire and the other with the first dependent's date of birth.

This example, while not obvious at first encounter, is worth some study: it shows how you can use Labeled Dependent USINGs to map very complex structures in a natural, readable way that does not require you to understand what pointers may have been established in which registers many lines earlier in the program.

Note that the DROP statement, by specifying the two qualifiers, removes the mappings of both **Date** DSECTs.

39.7.3. Personnel-File Record Example: Copying Addresses

Suppose our third requirement is to update the employee record so that the second dependent's address is made to be the same as that of the employee. In this case, the addresses are at the same level of nesting: the person's home address is nested within the **Person** DSECT at the position Labeled **PAddr**. This means that we must provide addressability to two different **Addr** DSECT instances, as shown in Figure 669.

```
* Example 3: Copy Employee Address to Dependent 2 address

AE   6   USING Addr,PE.PAddr       Overlay Addr DSECT on Employee name
                    1
PD2  7   USING Person,EDep2        Overlay Person DSECT on Dependent 2
AD2  8   USING Addr,PD2.PAddr      Overlay Addr DSECT on Dep. 2 Person
                    7

         MVC   AD2.AddrF,AE.AddrF  Copy Employee Addr to Dependent 2
                8        6

         DROP  PD2                 Remove all Dependent-2 associations
```

Figure 669. Copying addresses with an Employee Record

This technique is like that of the previous example: we establish addressability to the instances of the **Addr** DSECT within the two instances of the **Person** DSECT, one for the employee at **EPerson**, qualified with **PE** (1) in Figure 666, and one for the second dependent at **EDep2**, qualified with **PD2** (7).

Within the two instances of the **Person** DSECT are the two instances of the **Addr** DSECT, one for the employee at **PE.PAddr**, qualified by **AE** (6), and one for the second dependent at **PD2.PAddr** (7), qualified by **AD2** (8). The move instruction then uses the “address qualifiers” **AE** and **AD2** to qualify the names of the field to be moved, **AddrF**.

The DROP statement specifies qualifier **PD2**. Because the Labeled Dependent USING with the **AD2** qualifier was based (or “anchored”) on that with qualifier **PD2**, dropping **PD2** automatically causes **AD2** to be dropped also.

Exercises

39.7.1.(1)+ What are the lengths of the **Date, Addr**, and **Phone** DSECTs?

39.7.2.(2)+ Using the values you found in Exercise 39.7.1, find the lengths of the **Person** and **Employee** DSECTs.

39.7.3.(2)+ Assemble a program containing the five DSECTs used in the personnel record and the statements in Figures 664, 667, 668, and 669. Study the generated object code to see how the single ordinary USING statement provides a base register for all the instructions.

39.7.4.(2)+ Create a definition of a 95-byte data structure named **Person** containing these items, where their lengths are shown in parentheses:

- Name, consisting of
  - Family name (20)
  - Given name (15)

  - Initial (1)
- Address, consisting of
  - Number (5)
  - Street name (18)
  - Apartment number (3)
  - City (15)
  - State (13)
  - Postal ZIP code (5)

1. First, define each field using EQU statements as an offset from the symbol **Person**.
2. Second, define each field with DS statements.
3. Third, define each field in a Dummy Control Section named **Person**.

39.7.5.(2)+ Suppose the last three statements in Figure 660 are replaced by

```
DateF    Equ   Date,DateL,C'C'
```

Will all symbols in the **Date** DSECT have identical attributes? (Write and assemble a short program to verify your answer.)

39.8. Summary

Enhancements to the USING and DROP statements can help you to write simpler and more efficient programs.

39.8.1. USING Statement Summary

The Assembler provides two major extensions to the USING statement: *Labeled* and *Dependent*. They may also be used in combination, as *Labeled Dependent* USINGs, providing a choice of four different types of USING statement.

This gives you much greater control over assignment and resolution of base addresses in symbolic expressions and helps improve the reliability, maintainability, and efficiency of Assembler Language applications.

- *Labeled USINGs* let you simultaneously address multiple instances of a DSECT (or CSECT) without additional ordinary USING and DROP statements, and without the need to explicitly code offsets and base registers. Thus, you can concurrently manage multiple copies of the same DSECT- or CSECT-defined data structure using the full symbolic capabilities of the Assembler Language.
- *Dependent USINGs* let you address multiple DSECTs anchored by a single base register, so you can describe adjacent, nested, or overlapping code and data structures. This means that you can reduce the number of general registers required for addressing DSECTs and assign them to other uses, creating more efficient code while retaining the advantages of fully symbolic addressing.

  Dependent USINGs are "dynamic" because declarations can specify different mappings on different code paths.
- *Labeled Dependent USINGs* combine the benefits of both. For example, you can describe record structures containing multiple instances of nested substructures, or of substructures that depend on a variable elsewhere in the containing structure. Such complex data structures are commonly used in higher level languages; you can use them in your Assembler Language programs.

Here's another way of viewing the difference between Labeled and unlabeled USING statements:

- An unlabeled USING requires a register to reference the data; or the base register implies the data that can be addressed.
- For a Labeled USING, the qualified data name implies the register: that is, the qualifier designates a specific base register.

The properties and behavior of the four types of USING statement are shown in Table 426:

USING Type	Label	Register Usage	Operand 1 Based on	Operand 2	Operand 2 Location in Storage	Number of Instances of Active Objects
Ordinary	No	1 register per object	Register	Absolute [0,15]	Anywhere in storage	Only one active instance of an object at a time
Labeled	Yes	1 register per object	Register	Absolute [0,15]	Anywhere in storage	As many active instances of an object as registers assigned
Dependent	No	Multiple objects per register	Operand 2	Relocatable, addressable	Addressable by ordinary USINGs	Multiple active objects of different types
Labeled Dependent	Yes	Multiple objects per register	Operand 2	Relocatable, addressable	Addressable by ordinary USINGs	Multiple different active objects of the same or different types

Table 426. Summary of USING Statements

39.8.2. DROP Statement Summary

The DROP statement supports the enhancements to the USING statement, as summarized in Table 426:

USING Type	DROP Statement
Ordinary	By register number
Labeled	By qualifying label (dropping the register has no effect)
Dependent	By register number (all sub-dependent USINGs are dropped automatically)
Labeled Dependent	By qualifying label (dropping the register has no effect)

Table 427. Summary of DROP Statement Behaviors

These may be described as follows:

- Ordinary USINGs

  The normal rules for DROP statements apply, and the entry for the specified register is removed from the Assembler's Using Table.

- Labeled USINGs

  The qualifying label from a previous Labeled USING is specified as the operand of the DROP statement. Only the USING with that qualifier is inactivated; other USINGs specifying the same base register (if any) are still active.

- Dependent USINGs

  The syntax of the ordinary DROP statement is used: a register is specified as the operand. Any further Dependent USINGs based on the same register are automatically dropped at the same time. The Assembler's Using Table entry for that register is removed.

- Labeled Dependent USINGs

  The qualifying label from a previous Labeled or Labeled Dependent USING is specified as the operand of the DROP statement. Any dependent or Labeled Dependent USINGs that relied on the qualifying label are also dropped. Other USINGs specifying the same base register (if any) are still active.

Terms and Definitions

Dependent USING
A USING statement allowing implicit references to symbols in areas mapped by more than one control section to be resolved with a single base register.

DSECT
(1) An assembler instruction defining the start or continuation of a Dummy Control Section. (2) A Dummy Control Section used to map the components of a data structure.

Labeled Dependent USING
A USING statement allowing implicit references to symbols in areas mapped by more than one control section to be resolved by a specific base register.

Labeled USING
A USING statement directing resolution of implicit references to a specific base register. It is distinguished from an ordinary USING statement by the presence of a qualifier in the name field.

Ordinary USING
A USING statement directing resolution of unqualified implicit references to a specific base register or registers.

qualified symbol
An ordinary symbol prefixed by a *qualifier* and separated from it by a period, as in `qualifier.symbol`.

qualifier
A symbolic identifier defined in the name field of a Labeled USING or Labeled Dependent USING statement. It may be used only as a qualifier, and not as an ordinary symbol.

40. Basic Data Structures

```
         44     00000000
        444    0000000000
       4444   00        00
      44 44   00        00
     44  44   00        00
    44   44   00        00
   44444444444 00        00
  444444444444 00        00
         44   00        00
         44   00        00
         44    0000000000
         44     00000000
```

In this section we discuss several widely used data structures such as arrays, stacks, lists, queues, trees, and hash tables.

The data structures used in computer programs are incredibly varied. For most purposes, understanding the simpler forms provides a basis for understanding all of the more complex ones.

- The simplest data structure is the *table* or *array,* a grouping of elements of uniform type and length into regularly spaced (and usually contiguous) memory locations. The major advantage of the array is that the address of an array element can be calculated and the element accessed directly, rather than searching the array to find the element.
- A *stack* is a specialized form of array, often represented in memory in varied ways.

Other data structures involve multiple elements of (usually) uniform type and length that are *not* contiguous or regularly spaced. Dummy Control Sections ("DSECTs") and extended USING statements provide powerful mechanisms for describing, mapping, and manipulating them.

- A *list* or *linked list* is a linearly ordered structure in which each element contains a link to its successor.
- A *queue* or *doubly linked list* is a linearly ordered structure in which each element contains links to its successor and to its predecessor.
- The elements of a *tree* contain links to zero, one, two, or more successors.

Lists and trees can be used to store data in forms not well adapted to arrays, but their elements must be searched for, instead of addressed directly.

- A *hash table* provides a method for rapid retrieval of unordered data, by transforming an item into a "pseudo-subscript" to locate its value quickly.

We will examine each of these basic structures in turn.[285] Because we often arrange data in linear, rectangular, or other array-like tabular formats, we will first examine some methods used to manipulate data stored in this form.

---

[285] The classic reference for data structures is Chapter 2 of *The Art of Computer Programming*, Volume 2: *Fundamental Algorithms*, by Donald Knuth.

40.1. One-Dimensional Arrays

Many examples of loops in Section 22 involved indexing in an array, but we didn't describe formal techniques for doing indexing arithmetic and selecting initial and final index values.

The most important new concept concerning array manipulation is that we may compute the position of an element at *execution* time. Up to now we have discussed many basic assembly-time addressing expressions; now we consider instruction sequences that evaluate more complex execution-time addressing expressions.

Suppose you have eleven halfword integers to manage as a group; the simplest arrangement is successive halfwords in memory. Here is an array of halfwords, starting at the halfword named **B**.

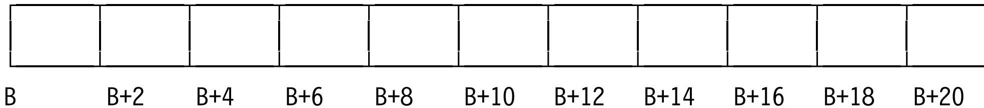


Figure 670. Example of a one-dimensional array of halfwords

One-dimensional arrays are relatively simple: each successive data item or element is accessed by adding the element spacing (usually, the element length) to the address of the preceding element. If for example the eleven halfword integers in Figure 670 are stored starting at **B**, then the n-th element B_n is found at B+2n. That is, if the element with zero subscript (B_0) is found at **B**, the address of an element with subscript n is simply B+2*n. If the array elements were fullwords or doublewords, the corresponding addresses would be B+4*n and B+8*n respectively.

However, arrays may not contain a “zero-th” element, or the zero-th element may not be the lowest-numbered element. For programs in many higher-level languages, the first element of an array has subscript 1. Indeed, if the three low-numbered elements of the array are omitted, so that B_4, B_5, ..., B_{14} are stored beginning at **B**, and the length of a single array element is L, then the n-th element B_n is found at address `addr(B)+L*(n−4)`.

We can see how to do the required subscript arithmetic: if the subscripts start at m, and the lowest-subscripted element B_m is stored at **B**, then the address of element B_n is

```
addr(B(n)) = addr(B)+L*(n-m).
```

An example will help to illustrate this. Suppose an array of 13 fullword integers X_5, X_6, ..., X_{17} is stored beginning at **XX**, and we want to store their sum at **XSum**. Thus, the array starts with the element with subscript 5. In Figure 671, we assume that the lower and upper subscript bounds of 5 and 17 are stored at **LowerSub** and **UpperSub** respectively. Because there are 13 elements in the array starting at **XX**, we could simply add the 13 words. However, we want to illustrate a more general technique for accessing array elements.

```
         SR    0,0              Initialize sum in GR0 to zero
         L     1,LowerSub       Initialize subscript n to c(Lower)
SetNdx   LR    2,1              Index N is calculated in GR2
         S     2,LowerSub       (n-m)
         SLL   2,2              4*(n-m): element length is 4
         A     0,XX(2)          Sum = Sum + X(n)
         LA    1,1(,1)          Increment subscript by 1
         C     1,UpperSub       Compare subscript to upper bound
         JNH   SetNdx           If not greater, branch to repeat
         ST    0,XSum           Store sum
         - - -
LowerSub DC    F'5'             Lower subscript bound
UpperSub DC    F'17'            Upper subscript bound
XSum     DS    F                Space for sum
XX       DC    13F'12345'       ... for example
```

Figure 671. Sum of array elements with known subscript bounds

Now, suppose the lower and upper subscript bounds of the elements used in forming the sum don't have *known* values like 5 and 17; we only know that the element with subscript 5 (X_5) is stored at **XX**. We can then include a portion of the indexing arithmetic into the program at *assembly* time. We extract the factor L*(−m) from the implied address XX+L*(n-m); m has value 5 in this example, and L has value 4.

```
         SR    0,0               Initialize sum to zero
         LA    4,L'XX            Increment = element length = 4
         L     2,LowerSub        Get initial (lowest) subscript
         SLL   2,2               Multiply by element length, 4
         L     5,UpperSub        Get highest subscript
         SLL   5,2               Multiply by length, = 4 also
AddElem  A     0,XX-5*4(2)       Add elements starting at X(5)
         JXLE  2,4,AddElem       Increment index and loop
         ST    0,XSum            ... etc.
```

Figure 672. Sum of array elements with unknown subscript bounds

In this more-efficient example, we see that if the contents of **LowerSub** and **UpperSub** are 5 and 17 respectively, the same result will be obtained as before. The address of the first element to be added will be XX-20+(4*5), which is the same as XX. The Assembler requires that the location of the implied address XX-5*4 be addressable; this requirement is sometimes a limitation on the use of this time-saving technique.

The loop in this program segment needs only two instructions, while the loop in Figure 671 on page 908 uses seven. You can often improve the efficiency of array accesses by doing as much subscripting arithmetic as possible at assembly time.

Even though we usually store arrays with ascending subscripts, the array elements could also be stored in “reverse” order, from highest subscript to lowest: you can still calculate the address of B(n) this way, even if n is not greater than m.

Exercises

40.1.1.(2) Suppose the instructions in Figure 671 on page 908 had been written so that the lower and upper subscript bounds were defined instead by the statements

```
LBound   Equ   5
HBound   Equ   17
```

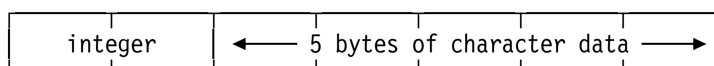
Rewrite the three DC statements so that all values depending on the subscript bounds are defined through references to the symbols **LBound** and **HBound**.

40.1.2.(2)+ A programmer suggested the following technique as a method of indexing through a table of words:

```
         - - -                   Initialize, etc.
GetIt    L     0,Table           Get a table entry
         - - -                   Do something with it
         LH    1,GetIt+2         Fetch the addressing halfword
         AH    1,=H'4'           Add 4 to displacement
         STH   1,GetIt+2         Replace halfword in instruction
         - - -                   Eventually branch back to 'GetIt'
```

Assuming that this code sequence contains proper tests for reaching the end of the table, are there any grounds (other than aesthetic) for criticizing this method of address modification?

40.1.3.(2) An array contains 523 elements. Each element is 7 bytes long: a 2-byte binary integer followed by 5 characters.



Give reasons for or against spacing successive elements of the array 8 bytes apart.

40.1.4.(2) In the example in Figure 672, what will happen if c(**UpperSub**) is less than c(**LowerSub**)?

40.1.5.(2)+ Each element of the array of 523 elements in Exercise 40.1.3 should be mapped by a DSECT; create one.

40.2. Two-Dimensional Arrays

Two- and higher-dimensional arrays present added considerations, but they can be handled easily; we will examine two simple methods for addressing array elements. In illustrating these methods, we discuss only the two-dimensional array, or *matrix.* Figure 673 illustrates a small matrix:

A(1,1)	A(1,2)	A(1,3)	Row 1
A(2,1)	A(2,2)	A(2,3)	Row 2
Column 1	Column 2	Column 3	

Figure 673. Typical arrangement of elements of a matrix

The element $A_{i,j}$ in the i-th row and j-th column of a matrix A is often denoted A(i,j). The row subscript is traditionally given first, followed by the column subscript.

The basic problem is that we rarely know the values of the subscripts i and j at assembly time. Instead, i and j are *variables* whose values are determined at execution time, so we must write instructions using those values to find the address of the array element A(i,j).

First, we must reorganize this rectangular array form into a linear structure conforming to the machine's linear addressing of memory. This can be done in two ways:

- Store successive *columns* of the array one after another. This is called "column order" or "column-major order", and is used, for example, by Fortran for storing its arrays. If the column-ordered array A in Figure 673 has two rows and three columns, it would be stored as shown in Figure 674. Note that the leftmost subscript is "cycled" most rapidly, so the linear order of the elements is first column, then second column, etc.

A(1,1)	A(2,1)	A(1,2)	A(2,2)	A(1,3)	A(2,3)

Figure 674. Storing an array in column order

- Store successive *rows* of the array one after another. This is called "row order" or "row-major order", and is used, for example, by PL/I for storing its arrays. The same array stored in row order would appear in storage as shown in Figure 675. Note that the rightmost subscript is "cycled" most rapidly, so the linear order of the elements is first row, then second row, etc.

A(1,1)	A(1,2)	A(1,3)	A(2,1)	A(2,2)	A(2,3)

Figure 675. Storing an array in row order

Either linear arrangement may be used; a choice between possibilities must be based on considerations such as convenience, the time and space required to retrieve a particular element, and (if your array-handling instructions will be called from a program written in another language) the requirements of that programming language.[286]

[286] Many programming language standards require that arrays be stored in a specific ordering of rows and columns.

To retrieve element A(i,j) of the column-ordered array in Figure 674, we must calculate its address: suppose A(1,1) is stored at **AA**. To obtain the address of the first element in the j-th column, A(1,j), we need the address

```
addr(AA) + L * (j-1) * 2
```

where L is the element length in bytes, and the factor of 2 is the number of rows, accounting for the presence of two elements in each column. Having obtained that address, the address of the i-th element in the indicated column is found by adding `L*(i-1)` to the partially computed address, giving finally an address

```
addr(A(i,j)) = addr(AA) + L * [ (i-1)+2*(j-1) ]
```

For example, if the elements of the array are halfwords (L=2), then the address of element A(2,3) is

```
addr(A(i,j)) = addr(AA) + 2 * [ (2-1)+2*(3-1) ]
             = addr(AA) + 10
```

as we would expect from Figure 674 on page 910.

The quantity added to addr(AA) is sometimes called a *subscripting expression,* a *subscripting function,* or a *mapping function*. We start with the address of the base element A(1,1) and the array subscripts i and j of a particular element, and use them to compute the *linear subscript* that gives the difference between the addresses of A(i,j) and A(1,1).

If a two-dimensional column-ordered array has *r* rows, the subscripting function is

```
L * [ (i-1)+r*(j-1) ]
```

Our choice of column ordering means that the subscripting function depends only on the number of rows *r* of the array, and not on the number of columns.

Suppose a column-ordered array of words having 5 rows and 7 columns is stored with base element A(1,1) at **AA**. We want to retrieve element A(i,j) and store it at **AIJ**. The subscripts i and j are contained in fullwords stored at **ISub** and **JSub** respectively. We assume values have been stored in the array at **AA** by other parts of the program.

```
NRow     Equ   5                 Number of rows
NCol     Equ   7                 Number of columns
         L     6,JSub            Get column index j
         BCTR  6,0               Form (j-1)
         MHI   6,NRow            Multiply by number of rows
         A     6,ISub            Add row index, i
         BCTR  6,0               Decrease by 1 to form ((i-1)+r*(j-1))
         SLL   6,2               Multiply by element length, 4
         L     0,AA(6)           Retrieve element A(i,j)
         ST    0,AIJ             Store it at AIJ
         - - -
ISub     DC    F'3'              Possible value for i
JSub     DC    F'6'              Possible value for j
AIJ      DS    F                 Element A(i,j) stored here
AA       DS    (NRow*NCol)F      ... Values already placed here
```

Figure 676. Retrieving a specified element of an array

As in Figure 672 on page 909 for one-dimensional arrays, part of the execution-time subscripting arithmetic can be absorbed into the assembly-time address of the instruction that references the array element. To do this, we must know the size of the array (at least, the number of rows, for a column-ordered array) when the program is assembled. We now see that

```
addr(A(i,j)) = [addr(AA) - (L*(r+1))] + [L * (i + r*j)].
```

Only the second square-bracketed expression depends on i and j, so it must be computed at execution time. The instructions in Figure 676 can be rewritten as in Figure 677 on page 912:

```
NRow     Equu 5                    Number of rows
NCol     Equ  7                    Number of columns
         L    6,JSub               Column index
         MHI  6,NRow               * (Number of rows)
         A    6,ISub               + Row index
         SLL  6,2                  (All this) * (element length)
         L    0,AA-L'AA*(NRow+1)(6)   c(GR0) = A(i,j)
         ST   0,AIJ                Store result at AIJ
```

Figure 677. Retrieving a specified element of an array efficiently

The address `addr(AA)-L*(r+1)` is the address of the "element" A(0,0), which may not actually be a member of the array.[287] The term "virtual origin" describes the base element A(0,0), and the term "actual origin" describes the address in memory where the array actually begins.

Most programming languages that support arrays allow many more than two dimensions, but the terms "row" and "column" ordering are still used for higher-dimension arrays, even though these names may seem misleading. (Should the third dimension be called a plane? The fourth dimension a cube or a parallelepiped?) Whatever the terminology, remember that the leftmost subscript varies fastest for column ordering, and the rightmost for row ordering.

We will assume for now that all subscripts begin at 1, unless stated otherwise.

Exercises

40.2.1.(3) Suppose an N by N array of words is stored starting at **SqArray**, where N is greater than 6 and is defined symbolically. Write a sequence of statements that will *transpose* the array: that is, for all subscripts i and j, swap element SqArray(i,j) with element SqArray(j,i).

Do you need to know whether the array is stored in row order or column order?

How many swaps are needed?

40.2.2.(3)+ There are two symbolically defined integers M and N, both greater than 5, and $M \neq N$. Given an N by M matrix of fullwords stored at **AA**, store at **BB** the M by N *transpose matrix* of the matrix at **AA**, whose elements are defined by B(i,j) = A(j,i).

Do you need to know the ordering of the array?

40.2.3.(3)+ Suppose you are given the address of the actual origin of a column-ordered array, addr(A(1,1)). You also know the number of rows R and the number of columns C of the array, and the length L of each element. Suppose the word at **ElemAddr** contains the address of an unknown array element A(i,j).

First, write expressions that determine the values of i and j. Then, write instructions that calculate the subscripts i and j, and store them at **ISub** and **JSub** respectively.

40.2.4.(2) *Row-ordering* an array requires that the linear arrangement corresponds to cycling the rightmost subscript most rapidly, then the next to last, and the leftmost most slowly. Show that the subscripting function for a two-dimensional array of *r* rows and *c* columns is

```
addr(A(i,j)) = addr(A(1,1)) + L * [ c*(i-1)+(j-1) ].
```

where L is the byte length of each array element.

40.2.5.(4) A physicist needed a table of values of a function S(i,j) for integer values of i = 2, 4, 6, ..., 20, and j = −3, −2, −1, 0, ..., 10. He found that S(i,j) obeys the recurrence relations

```
(a) S(i,j+2) = ((j+1)*S(i,j)+1/(i+j+2))/(j+3)
(b) S(i+2,j) = S(i,j) − 1/((i+1)*(i+j+2)),
```

and that

[287] Unfortunately, it's sometimes called the "base address" of the array, which can be confused with the address where the array starts, or with a base address specified in a USING statement.

```
S(2,-3) = 0.5, S(2,-2) = 0.3068528, and S(2,-1) = 0.23370055.
```

Write instructions to compute the complete table of 140 values in hexadecimal or binary floating-point, applying first the relation (a) to compute S(2,j) for j = 0, 1, ..., 10, and then using (b) to fill out the table.

40.2.6.(3) We can define two "norms" of a matrix as (1) the sum of the magnitudes of its elements, and (2) as the magnitude of of the largest element. Write a program segment which will evaluate both norms of a square matrix of short hexadecimal or binary floating-point numbers stored beginning at **Matrix**, and the dimension of the matrix is stored as a fullword integer at **MatDim**.

40.3. General Array Subscripts

In the examples above we assumed that the subscripts took positive values only, and always had a lower bound of 1. (This is why our subscripting expressions contain terms like (i-1) and (j-1).) This is not a necessary condition; if the lower subscript bounds on i and j are i0 and j0 respectively, then the subscripting function for a two-dimensional column-ordered array with *r* rows becomes

```
L * [ (i-i0)+r*(j-j0) ]
```

and for a two-dimensional row-ordered array with *c* columns,

```
L * [ c*(i-i0)+(j-j0) ]
```

In such cases it may be more difficult to include the constant factor that determines the array's "virtual origin",

```
-(L*(i0+r*j0))    or    -(L*(c*i0)+j0))
```

in an expression at assembly time, since the resulting implied address may not be addressable.

Many high-level programming languages let you define arrays dynamically, so that subscript bounds and origins are not known until execution time. References to such arrays customarily use *array descriptors* containing the necessary information.

Arrays can take many forms. We will examine two: arrays of many dimensions, and arrays whose columns are not identical.

40.3.1. Multi-Dimensional Arrays (*)

When you use arrays with more than two dimensions, you will need to calculate the linear index of an element. For example, suppose D is the base element of a three-dimensional column-ordered array with R rows, C columns, and P "planes" (for want of a better name for the third dimension). Then to access element D(i,j,k) you can generalize the expression given in Section 40.2 as follows:

```
Addr(D(i,j,k)) = addr(D) + L * [ (i-1) + R*[(j-1) + C*(k-1)] ]
```

To help understand this expression, consider the element D(1,1,2): to access it starting at D(1,1,1) you would cycle the i subscript R times, and the j subscript C times, once for each cycle of the i subscript. Thus,

```
Addr(D(1,1,2)) = Addr(D) + L * [ (1-1) + R*(1-1) + R*C*(2-1) ]
               = Addr(D) + L * (R*C)
```

where R*C is the number of elements in the first "plane", so the accessed element will be the first one in the second "plane" as desired.

In general, if an array A has p dimensions, and the subscripts k_1, k_2, ..., k_p in each dimension range from 1 to Max_j (where "j" is the number of subscript k_j), then the subscripting function for an element A(k_1,k_2,k_3, ..., k_p) is

$$addr(A(k_1,k_2,k_3,...,k_p)) = addr(A(1,1,1,...,1)) + L * [(k_1-1) + (k_2-1)*Max_1 + (k_3-1)*Max_1*Max_2 + ... (k_p-1)*Max_1*Max_2*...*Max_{(p-1)}]$$

Not obvious, perhaps, but sometimes useful to know!

As we noted in describing subscripts for two-dimensional arrays, the range of values for the rightmost subscript in a column-ordered array isn't used in the subscripting function. That is, for a two-dimensional array, only the number of rows us important, not the number of columns. For a row-ordered array, the range of the leftmost subscript is not used in the subscripting function.

40.3.2. Non-Homogeneous Arrays (Tables)

The term “table” often describes a one-dimensional array in which the columns don't necessarily contain data items with the same type and length. For example, you might have a table of names and identification numbers, where the numbers in the first column are 4-byte binary integers and the names in the second column are 36-byte character strings.

ID Number	Name
14142135	Billy Uss
17320508	Norm D. Plume
16180339	Anna Lepsis
28571428	Carol F. DeBelz
31415926	Warren Pease
33550336	Sarah Bellum
22360679	Pete Moss
40353607	Polly Andress
31622776	Hugo Furst
24137569	Lou Gubrius
16777215	Anna Lemma
27182818	Dad Gummitt
11235813	Polly Connick

Table 428. Example of a non-homogeneous array

The methods for indexing one-dimensional arrays are used to access the rows of such tables; the increment from row to row is the length of each row (40 in this example).

For example, suppose this table starts at **DataTbl** and there is an ID number at **WhoIsIt**. You want to find the table entry corresponding to that ID number, and put its address in GR1. A typical code fragment might look like Figure 678. (Note that it doesn't test whether all table entries have been compared, to indicate there's no matching entry.)

```
EntryLen Equ   40                 Each table entry is 40 bytes long
         LA    1,DataTbl          c(GR1) = start of data table
         L     0,WhoIsIt          c(GR0) = IDNumber being sought
Search   C     0,0(,1)            Check for a matching ID number
         JE    FoundIt            They're equal, we have a match
         LA    1,EntryLen(,1)     Add length of each table entry
         J     Search             And repeat the search
FoundIt  - - -
```

Figure 678. Searching for a matching table entry

Each element in Table 428 might be described by this DSECT:

```
ID_Name  DSECT ,                  Dummy Control Section
IDNumber DS    F                  4-byte integer ID number
Name     DS    CL36               36-character name
```

Because DSECTs generate no object code, you could also write the DSECT with DC statements:

```
ID_Name  DSECT ,                  Dummy Control Section
IDNumber DC    F'-1'              4-byte dummy integer ID number
Name     DC    CL36'Dummy Entry'  36-character dummy name
```

The result is the same in both cases. Using DC statements can be helpful in cases where the same statements can be included[288] in both a CSECT for a declaration and in DSECTs for references.

We can rewrite the example in Figure 678 on page 914 to use the DSECT this way:

```
         LA    1,DataTbl         c(GR1) = start of data table
         Using ID_Name,1         Base reg for ID_Name data structure
         L     0,WhoIsIt         c(GR0) = IDNumber being sought
Search   C     0,IDNumber        Check for a matching ID number
         JE    FoundIt           They're equal, we have a match
         LA    1,EntryLen(,1)    Add length of each table entry
         J     Search            And repeat the search
FoundIt  - - -
         - - -
ID_Name  DSECT
IDNumber DS    F                 4-byte integer ID number
Name     DS    CL36              36-character name
EntryLen Equ   *-ID_Name         Calculate length of each table entry
```

Figure 679. Searching for a table entry mapped by a DSECT

The second USING statement tells the Assembler that implicit references to any symbol in the DSECT ID_Name should be resolved using GR1 as a base register. The USING Table is sketched in Figure 680:

basereg	base location	RA
15	00000002	01
1	00000000	FF ←— Relocation ID for the DSECT

Figure 680. USING Table with two entries, one for a DSECT

Important differences between Figures 678 on page 914 and 679 include:

- The instruction named **Search** makes no explicit reference to GR1, but relies on the Assembler to calculate the correct base and displacement.
- The second operand is the *name* of the field, and not its (known) offset in the table entry.

Advantages of this technique include:

- Readability: the referenced field is identified by name, so you need not know its offset within the table entry mapped by the DSECT.
- If other data elements are later added to the table entries in the table, only the DSECT needs to be changed. If explicit base and displacements had been used to reference components of a table entry, each would have to be found and modified.
- The length of the table entry is calculated by the Assembler from information it knows about the length of the DSECT.
- If new components are added to the table entry, the Equ statement in Figure 678 on page 914 isn't needed, because the symbol **EntryLen** is automatically updated by the Assembler.

[288] The COPY assembler instruction statement inserts a block of statements in its place.

Exercises

40.3.1.(2) Do the same as in Exercise 40.2.2, but use no memory locations for temporary storage. A simple solution will be satisfactory! "Optimal" solutions to this problem can be rather complex.

40.3.2.(3)+ Given two N by N square matrices of fullword integers stored in column order at **AA** and **BB** respectively, the "product matrix" CC is defined by

```
C(i,j) = Sum( for k=1 to N ) [ (A(i,k) × B(k,j)) ]
```

Assuming that N is defined symbolically and is greater than 4, write statements to store at **CC** in column order the product matrix whose elements are given by the above rules for a product matrix. Assume that no overflows occur in calculating C(i,j).

40.3.3.(2) Write the subscripting function for a p-dimensional row-ordered array.

40.3.4.(3)+ It often occurs in processing square matrices that a matrix is *symmetric*: that is, A(i,j) = A(j,i) for all subscripts i and j. To save space, such matrices are usually stored in either

- *upper triangular* form, where A(i,j) is not stored if (i>j); or
- *lower triangular* form, where (i,j) is not stored if (i<j).

This saves roughly half the space needed to store the matrix.

For an N by N matrix of elements of length L, determine the subscripting function needed to retrieve element A(i,j) for each type of triangular matrix.

40.3.5.(3)+ Using your solution to Exercise 40.3.4, write instructions that place in GR1 the address of the element A(i,j) of an upper triangular matrix, given that A(1,1) is stored at **AA**, and the quantities i, j, N, and L are fullword integers stored at **II**, **JJ**, **NN**, and **LL** respectively. Assume that the subscript values are valid.

40.3.6.(4) Suppose you have an n-dimensional column-ordered array for which the subscripts range from 1 to Upper_k for each dimension k between 1 and n. Write an instruction sequence that calculates the linear subscript of an arbitrary element, assuming:

- The (1,1,...,1) element of the array is stored at **ARRAY**.
- The subscripts from 1 to k of the desired element are fullword integers stored in an array at **SubScrs**.
- The upper limits Upper_k for each subscript are word integers stored in an array at **UpperLim**.
- The number of dimensions is stored in the word at **NumDimen**.
- The length of each array element is stored in the halfword integer at **ElLength**.

40.3.7.(2)+ In some applications involving large matrices, there may be few nonzero elements; they are called "sparse" arrays. Instead of allocating storage for all N^2 elements of an N×N matrix, space can be saved by storing only the nonzero elements A(I,J) and their subscripts I and J.

Suppose there are three linear arrays representing a sparse array A: **IVal** and **JVal** contain the word subscripts I and J of the element stored at **AVal**, and the number of elements in each linear array is stored in the word at **NbrEls**. Write an instruction sequence that forms the *transpose* of the matrix A. That is, for each element A(I,J), swap I and J.

40.3.8.(0) For fun: deduce the meanings of the ID Numbers in Table 428 on page 914.

40.4. Address Tables

A second method of array addressing is useful when processing speed is important, and occasionally finds application to arrays of irregularly-spaced or irregular-length data. We precompute the addresses of portions of the array, and store those addresses in a separate *address table* or *access table.* For example, suppose the addresses of the elements A(1,1), A(1,2), and A(1,3) in Figure 673 on page 910 are stored as words in a list beginning at **ColAddr**, as shown in Table 429.

Location	Contents
ColAddr	addr(A(1,1))
ColAddr+4	addr(A(1,2))
ColAddr+8	addr(A(1,3))

Table 429. Array addressing with a table of addresses

The following example uses this table to retrieve element A(i,j) and store it at **AIJ**.

```
         L     7,JJ              Get column index j from JJ
         BCTR  7,0               Decrease by 1 for indexing
         SLL   7,2               Multiply by address length, 4
         L     6,ColAddr(7)      Get address of column j
         L     5,II              Get row index i from II
         BCTR  5,0               Decrease by 1 for indexing
         SLL   5,2               Multiply by element length, 4
         L     0,0(5,6)          Get A(i,j) from array
         ST    0,AIJ             Store at AIJ
```

Figure 681. Creating a table of addresses

An advantage of this scheme is that it avoids the previously required multiplication by the number of rows. The added expense is (1) the space required for the table, (2) the time required for forming it, either during assembly or at execution time, and (3) the cost of an additional memory access.

As a final example, suppose we want to store at **AIJ** the element A(i,j) of a 5-by-5 array of fullwords stored in column order at **AA**. First, we construct a table of column addresses and store them at **AddrTab**. If we actually compute the addresses of the first element in each column minus 4 (the length of each element), we can use the subscript “i” directly without subtracting 1 when accessing the desired array element.

```
NRows    Equ   5                 Number of rows
NCols    Equ   5                 Number of columns
         LA    6,NRows           c(GR6) = number of rows
         SLL   6,2               Multiply by element length
         LA    5,NCols           c(GR5) = No. Columns = loop count
         LA    9,AddrTab         Beginning address of table
         LA    0,AA-4            (array address)-(element length)
StoreAdr ST    0,0(0,9)          Store an address in the table
         AR    0,6               Increase to address of next column
         LA    9,4(0,9)          Increase table address to next word
         JCT   5,StoreAdr        Loop until all addresses computed
         - - -
AddrTab  DC    (NCols)F          Space for column addresses
```

Figure 682. Creating a better table of addresses

Table 430 on page 918 shows the table's contents after executing this code segment. The zero subscript in the elements A(0,j) indicates that one element length has been subtracted from the address of the beginning of the j-th column of the array.

Location	Contents	Element Addressed
AddrTab	addr(AA-4)	A(0,1)
AddrTab+4	addr(AA-4+20)	A(0,2)
AddrTab+8	addr(AA-4+40)	A(0,3)
AddrTab+12	addr(AA-4+60)	A(0,4)
AddrTab+16	addr(AA-4+80)	A(0,5)

Table 430. Example of an address table's contents

Now, we can use this table to retrieve the desired element:

```
         L     2,II              Get row index i from II
         L     3,JJ              Get column index j from JJ
         SLDL  2,2               Multiply both indexes by 4
         L     3,AddrTab-4(3)    Get column address from table
         L     0,0(2,3)          Retrieve the element A(i,j)
         ST    0,AIJ             Store it at AIJ.
```

This example gives much faster access to the desired array element. It also uses the SLDL instruction to take advantage of the fact that the array elements and the entries in the table of addresses have the same length, which might not be true in general.

The address table can be constructed by the Assembler if the array dimensions are known. The items in the middle column of Table 430 can be used as operands in DC statements, because the expression in an A-type constant can be relocatable. Thus, we can generate the same address table at assembly time.

```
NRows    Equ   5                        Number of rows
L        Equ   4                        Length of array element
AddrTab  DC    A(AA-L)                  Addr(first column) - L
         DC    A(AA+L*(NRows)-L)        Addr(second column) - L
         DC    A(AA+L*(NRows*2)-L)      Addr(third column) - L
         DC    A(AA+L*(NRows*3)-L)      Addr(fourth column) - L
         DC    A(AA+L*(NRows*4)-L)      Addr(fifth column) - L
```

Figure 683. Creating a table of addresses at assembly time

The expressions in the address constants are written so that you need only specify the value of **NRows** in the first EQU statement, and the required addresses will be calculated by the Assembler.

The extension of address tables to higher-dimensioned arrays is straightforward. Each subscript is used in turn to retrieve the address of the next lower-level address table, and the next-to-last subscript accesses a table like the one in Figure 683. The final subscript then accesses the desired data element. This method is useful when the array cannot be kept entirely in storage, since only the top-level table and some “availability flags” need be retained; the lower-level tables and the array itself can be kept on secondary storage.

Exercises

40.4.1.(2) Create an address table (or address tables) for a three-dimensional array starting at **Box** having 3 rows, 4 columns, and 5 “planes”. Thus, the array element with highest subscripts is Box(3,4,5).

40.4.2.(2)+ Construct an address table for a three-dimensional array A that has maximum subscripts of 4, 7, and 5 for the first, second, and third subscripts, respectively. How do the entries in the address tables depend on the choice of row-ordering or column-ordering for the array?

40.4.3.(3) Improve the coding of Table 429 on page 917 and Figure 681 on page 917 by including factors of `L*(−m)` in the statements wherever possible.

40.4.4.(4) Given an N by M array P of fullword integers stored in column order at **PP**, replace each element by the average of its four “nearest neighbors”. That is, calculate

```
P(i,j) = [(P(i+1,j) + P(i-1,j) + P(i,j+1) + P(i,j-1))]/4
```

For the “edge” elements, assume that the missing term (or terms) has value zero.

40.4.5.(3)+ A two-dimensional array of fullword integers is stored in column order with element X(1,1) at **XX**. The lower and upper bounds for the subscripts in each dimension are word integers stored at **Lower1** and **Upper1**, and at **Lower2** and **Upper2**, respectively. Write a program segment to load into GR9 the element of the array whose subscripts are the word integers at **Sub1** and **Sub2** respectively. That is, the desired element is `X(c(Sub1),c(Sub2))`.

If the subscripts lie outside the lower or upper bounds, branch to **SubsErr**.

40.4.6.(2)+ Generate the address table in Figure 683 on page 918 with a single DC statement, retaining the symbolic definitions of **NRows** and **L**.

40.4.7.(3) The instructions following Table 430 on page 918 show how you can use an address table to access an array element without lengthy subscript calculations. Determine practical limits on the values of the subscripts i and j.

40.5. Searching an Ordered Array

We must often search an ordered array to find a match for a given value (called the *key* or *search argument*). Once found, that array element may be modified; more often, its index is used to retrieve associated data in other columns, or in similarly ordered arrays.

For a short array (of perhaps 10 or less elements), it is simplest to search the array linearly. To illustrate, suppose an array of word integers at **AA** is arranged in order of increasing values, and GR2 contains a positive integer. If a match is found between the search argument in GR2 and one of the array elements, branch to **Found** with GR9 containing the offset of the matching element; if no match occurs, branch to **NoMatch**.

```
N        Equ   10                 Number of table entries
         LA    10,4               Increment = element length in GR10
         LA    11,4*(N-1)         Comparand = last element index in GR11
         SR    9,9                Initial index value = 0 in GR9
Test     C     2,AA(9)            Compare c(GR2) to array element
         JE    Found              Branch if a match occurs
         JXLE  9,10,Test          Increment index, try again
         J     NoMatch            No match found
```

These instructions make no use of the fact that the array at **AA** is ordered. If the value in GR2 matches none of the array elements, we always scan the entire array before branching to **NoMatch**. We rectify this oversight with one added instruction, as follows:

```
N        Equ   10                 Number of table entries
         LA    10,4               Increment = element length
         LA    11,4*(N-1)         Comparand = index of last element
         SR    9,9                Initial index = 0
Test     C     2,AA(9)            Compare c(GR2) to array element
         JL    NoMatch            No match if element is too small
         JE    Found              Branch if a match occurs
         JXLE  9,10,Test          Bump index by 4 and try again
         J     NoMatch            c(GR2) is too big to match
```

In this example, we will exit from the loop whenever we know the number in GR2 cannot match an array element. We might make use of the fact that taking the second branch to **NoMatch** implies that c(GR2) is greater than any array element.

A major weakness of this linear search technique becomes apparent as the length of the array increases. If we assume that GR2 contains one of the array elements chosen at random, then on the average we must compare it to half the array elements to find a match. If the array contains N elements, we will need an average of N/2 comparisons, which can become fairly expensive if the array contains (say) 2000 elements.

There are two good solutions to this problem of rapidly increasing search costs, *binary search* and *hashing*. Hash tables will be discussed in Section 40.10 on page 941.

For large *ordered* arrays, it is helpful to use the *binary search* or *binary chop* method.[289] Rather than scan through the array in a sequential fashion, we start at the middle. If the search argument (the "key") is less than that array element, we ignore the second half of the array, because all its elements must exceed the key; if the key is greater than the middle element, we ignore the first half of the array, because all its elements are smaller than the key.

Next, we compare the key to the middle element of the remaining *half* of the array. By "chopping off" half of the remaining array elements each time, we can greatly reduce the number of comparisons needed. Instead of the previous average of N/2 comparisons needed for a linear search, the binary chop requires a number close to $\log_2(N)$, the base 2 logarithm of N. For an array of 2000 elements, there will be at most 11 comparisons, instead of as many as 2000 and an average of 1000 for a linear search.

We illustrate a binary search in Figure 684 on page 921, using symbolically-defined registers and values. Its main feature is two registers symbolically named LP and HP that point to the low-addressed and high-addressed elements of the portion of the array **AA** being searched. The sequence of instructions beginning at **Setup** determines the address of a byte halfway between the low and high addresses. Because this midpoint might fall on an incorrect boundary, the NR instruction aligns the midpoint address properly.

[289] You can impress your mathematically inclined friends by calling it the "Bolzano-Weierstrass Algorithm".

```
L        Equu   4                    Length of an array element, which...
*                                    ...must be a power of two (for mask)
N        Equ   2000                 Number of array elements
SA       Equ   2                    Search argument register (key)
X        Equ   1                    Indexing register
HP       Equ   3                    High-address pointer register
LP       Equ   4                    Low-address pointer register
M        Equ   0                    Mask register (will contain -L)
*        Initialize
         LHI   M,-L                 -L used as mask and increment
         L     LP,=A(AA)            Initial low pointer
         L     HP,=A(AA+L*(N-1))    Initial high pointer
*        Set midpoint address
Setup    LR    X,HP                 Move high pointer, ...
         SR    X,LP                 ... - low pointer = portion size
         JM    NoMatch              No match if pointers have crossed
         SRL   X,1                  Halve size to get midpoint index
         NR    X,M                  Mask for proper alignment
*        Compare, and adjust the pointers
         C     SA,0(X,LP)           Compare to selected element
         JE    Matched              If equal, found a matching element
         JL    Lower                Branch if below midpoint
         LAY   LP,L(X,LP)           Move low pointer upwards
         J     Setup                And test for finish
Lower    LAY   HP,-L(X,LP)          Move high pointer downwards, ...
         J     Setup                And test for finish
*        Construct address of the item we found
Matched  LA    9,0(X,LP)            Set addr of matched element in GR9
         J     Found                And go away happy
```

Figure 684. Example of a binary search

The last two instructions place the address (rather than the index) of the matching element into GR9.

There are many ways to *sort* arrays of data items, some very simple and some quite complex.[290] For small arrays, you can use a simple "exchange" sort:

- Take the first element and compare it to the second. If the first is larger, exchange them.
- Now, compare the second and third, and again exchange them if the second is larger than the third.
- Continue this way until the last two elements have been compared (and possibly exchanged). The largest element will now be in the last element of the array.
- Start again with the first element, but continue only to the next to last item; this will become the second-largest item in the array.
- Repeat these steps until you have compared and exchanged only the first two elements; the first element will then be the smallest in the array.

The number of comparisons in an exchange sort is proportional to the square of the number of elements, so it should be used only for very small arrays.

[290] The classic reference for sorting and searching is *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, by Donald Knuth.

Exercises

40.5.1.(1) If we must search through an entire two-dimensioned array for some value, is it preferable for the array to be stored in row order or in column order?

40.5.2.(2) Suppose the array AA in Figure 684 on page 921 was ordered in *descending* order (the wrong way). What will happen?

40.5.3.(3) In Figure 684 on page 921 we assumed that the byte length of an array element was a power of two: in this case, 4. Rewrite the code sequence assuming that the length of an array element is contained in the word at **ElemLen**. To simplify, assume also that the part of the element to be matched will be fullword-aligned at the start of the element.

40.5.4.(2) Suppose an ordered array contains both positive and negative values. Write a code sequence that takes advantage of this fact by searching the array sequentially in a forward direction if the search argument is negative, and in a backward direction if it is positive.

40.5.5.(3) Assume the same ordered array as in Exercise 40.5.4. Write instructions that take advantage of this fact to improve a binary search. What improvement do you expect?

40.5.6.(3)+ Given an array of N word integers stored at **Data**, write instructions to sort the integers using an exchange sort.

40.5.7.(3) What will happen if you do a binary search for an element in an array that is not strictly ordered? Give an example.

40.5.8.(3) The discussion of an exchange sort above says that the number of comparisons is proportional to the square of the number of elements. Give a more precise estimate.

40.5.9.(3) Suppose we must search an ordered array at **Data** that contains 2000 elements, each 53 bytes in length. The first 30 characters of an array element hold a name, and the remaining 23 bytes contain related data. The 30-byte search argument to be sought in the array is at **SArg**. A branch to **Found** is to be made with the address of the matched element in R9, or to **None** if no match occurs. Write instructions to perform a binary search of the array, matching on the 30-character name.

40.5.10.(2) Compare the instructions you wrote in Exercise 40.5.9 to the following, and determine which is more efficient for an array of 2000 elements. Assume that matches are equally likely for all elements.

```
         LA    0,53              Element length, for compares
         L     3,=A(1024*53)     Initial increment = (El Length)*..
*  ..(Largest power of 2 contained in (number of elements))
         L     1,=A(Data)        Bottom address
         L     2,=A(Data+53*2000) Top address, after last element
         LA    9,0(1,3)          Set initial compare address
Test     CR    9,2               Check addr against top
         JNL   Low               If off top, subtract increment
         CR    9,1               Check addr against bottom
         JL    High              If off bottom, add increment
         CLC   SArg(30),0(9)     Compare search arg to element
         JE    Found             Branch out if that's it
         JH    High              Jump if arg bigger than element
Low      SR    9,3               Subtract current increment
         J     ChkEnd            Jump to test for end
High     AR    9,3               Add current increment
ChkEnd   CR    3,0               Compare increm to element length
         JL    None              If smaller, no match
         SRL   3,1               Halve increment
         J     Test              And try again
```

What modifications would be required to these instructions if we wanted to scan an array containing (1) 3 elements, (2) 2047 elements, (3) 2048 elements?

40.5.11.(3)+ In Figure 684 on page 921, the search argument is in a register and the array **AA** is a table of word integers. Revise the instructions in Figure 684 on page 921 assuming the array contains the *addresses* of character strings of length **SL** and your search argument is the address of a similar character string.

In this case, the strings can be anywhere in storage, but their addresses in the array **AA** refer to them in sort order.

40.6. Stacks

Unlike an array, which usually contains a fixed number of elements, stacks, linked lists, and trees usually contain varying numbers of elements. They are *dynamic* data structures, containing different numbers of elements as your program executes. Also, the organization of an array is tightly constrained, usually to a contiguous grouping of elements. (If it wasn't, we couldn't use efficient addressing expressions!) Stacks can be organized somewhat more loosely, while linked lists and trees may be quite disjointed in their arrangement in memory.

A stack, as its name implies, is a data structure to which elements are added, and from which elements are removed, at the "top". The requirement that data be added and removed at only one end is a characteristic property of a stack.

The terminology describing stack operations helps us visualize what is happening. When an element is added to the top of the stack, the other elements are "pushed down" one level; hence the name "push-down stack". Similarly, removing an element from the top allows the others to "pop up" one level.

40.6.1. An Example Using a Stack

Stacks can be used to convert expressions from their familiar *infix* form[291] such as 2+3*5, to a form more readily adapted to evaluation or (as in a compiler) code generation. For example, the expression 2+3*5 is by convention the same as 2+(3*5), because we assign a higher priority to multiplication than to addition. This form of expression is inconvenient for direct evaluation: if in scanning this "infix" expression from left to right, we might place the value 2 in a register and prepare to add it to something, we would then find that the "something" should have been evaluated first. It helps to have converted from the standard infix notation to an "operator suffix" or *postfix* form, but we won't show *how* to do that conversion here.[*]

To evaluate an expression in postfix form, two simple rules are used. When an *operand* is encountered in the left-to-right scan, its value is pushed onto the stack, and when an *operator* is encountered, the top two elements of the stack are removed, the operator is applied to them, and the result is pushed back onto the stack. For example, the infix expression 9*3+2 becomes 93*2+ in postfix form, and is evaluated this way:

1. Push 9 and then 3 onto the stack;
2. Remove them and apply the * operator, and place the result 27 back on the stack;
3. Push 2 onto the stack;
4. Remove the two values 27 and 2 from the stack, apply the + operator, and push the result 29 back onto the stack.

The stack now contains a single element, the value of the expression.

---

[291] Our traditional form of writing arithmetic expressions puts the operators between operands, with parentheses to indicate sub-expressions. Two forms more convenient for computer processing are the *prefix* and *postfix* notations. Both involve the use of stacks.

[*] Almost any textbook on compilers will show how it's done.

The infix expression 3+5*7 is represented in postfix notation as 357*+, so that all three operands are on the stack before the * operator is applied. The value of the expression will be (5*7)+3, or 38. If the infix expression had been written (3+5)*7, its postfix equivalent would be 35+7*, implying a very different order of evaluation: the operands 3 and 5 are stacked and then replaced by their sum 8; the operand 7 is pushed onto the stack, and then the * operator replaces the 8 and 7 by 56, the value of the expression.

40.6.2. An Example Implementing a Stack

The simplest implementation of a stack uses two items: a linear array of elements, and a "stack pointer" that may be either the subscript or the address of one of the array elements. By convention, the stack pointer locates the element on top of the stack. For example, Figure 685 shows a stack containing the values 9, 2, and 6. It might be represented in memory as the first three elements of an array of fullwords.

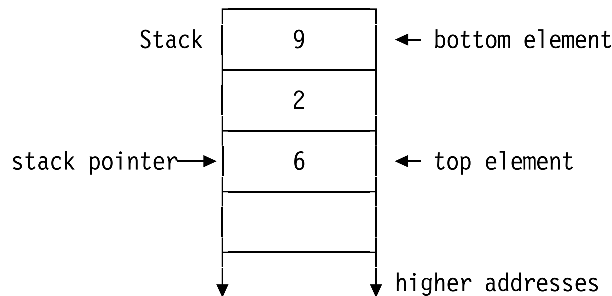


Figure 685. A stack growing toward higher addresses

The element 6 is at the top of the stack, and the element 9 is at the bottom.

We might implement this stack by allocating an array of 20 fullwords at **Stack**, and carrying the index of the top element in GR1.

```
SP       EQU   1                  GR1 contains stack pointer
StkSize  EQU   20                 Stack size (20 elements maximum)
         LM    2,4,=F'9,2,6'      Initialize with 3 elements
         STM   2,4,Stack          Put 3 items onto stack
         LA    SP,8               Set index of stack top to 3rd word
         - - -
Stack    DS    (StkSize)F         Allocate space for the stack
```

Figure 686. A stack implemented as an array

Now, suppose the number in GR0 is to be pushed onto the stack: we simply increment the stack pointer and store GR0 using the new index.

```
         AHI   SP,4               Increment stack pointer by 4
         ST    0,Stack(SP)        Store value at new top position
```

Figure 687. Pushing a data item onto a stack

If we think of the stack as an array, we have simply increased the subscript denoting the stack pointer by one; this is a common way to visualize a stack.

To remove an element from the stack, we need only move the stack pointer "down" one element, toward the bottom of the stack. Nothing need be done to the old top element, since it is now no longer a part of the stack.[292]

```
         AHI   SP,-4              Remove top element from the stack
```

[292] There may be situations where you *will* want to erase or remove the data left in deleted stack elements, to protect the data from being visible to other programs using the same stack (or peeking at activities in your program).

Now, suppose we want to replace the two elements at the top of the stack by their sum. We can do this by (1) popping the top two elements off the stack, (2) adding them, and (3) pushing the sum back onto the stack.

```
         L     0,Stack(SP)          Retrieve top stack element
         AHI   SP,-4                Now delete it from the stack
         A     0,Stack(SP)          Add the new top element
*        AHI   SP,-4           (**) Remove it from the stack
*        AHI   SP,+4           (**) Push the sum back on the stack
         ST    0,Stack(SP)          Store new top element
```

Figure 688. Adding top two elements of a stack

The second and third AHI instructions (marked "(**)" in the comment statements) aren't needed: their actions mutually cancel. Because we know that something will immediately go back onto the stack, it is safe to omit those two statements and leave the stack pointer momentarily referring to an element that was removed.

Because we may want to refer to several elements at the top of the stack, we can take advantage of the System z base-displacement addressing structure by placing the bottom of the stack at the highest-addressed element.

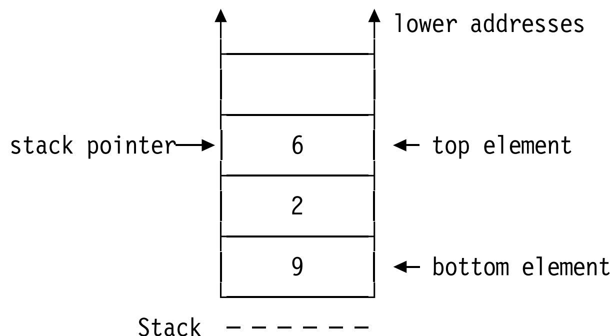


Figure 689. A stack growing toward lower addresses

The advantage of using instructions with nonnegative displacements may not be needed if you can access your stack elements using long-displacement instructions.

Pushing and popping are done as in Figure 688, except that the stack pointer now moves (arithmetically) in the opposite direction. We can refer to elements at and below the top of the stack using nonnegative displacements for instructions referring to the stack via the stack pointer. The example in Figure 690 evaluates the same sum as in Figure 688, except that the stack pointer now contains the *address* of the stack top rather than its index.

```
SP       EQU   1                    Register GR1 contains stack pointer
StkSize  EQU   20                   Stack size
         - - -                      Program puts data on stack
         L     0,0(,SP)             Get top stack element
         A     0,4(,SP)             Add next-to-top element
         ST    0,4(,SP)             Store sum at next-to-top position
         LA    SP,4(,SP)            Pop stack once to adjust
         - - -
         DS    (StkSize)F           Define space for stack elements
Stack    DS    0F                   Define stack name just past bottom
```

Figure 690. Add top two elements of a stack

Because we assigned the stack name to the first element position past the bottom of the stack, all legal values of the stack pointer in GR1 must be less than the address of **Stack**. We can use this property to test for the possibility of *stack underflow,* which occurs when too many elements have been removed from the stack. If the pointer is greater than or equal to the address of **Stack**, an underflow is indicated.

```
         LA    SP,4(,SP)          Pop an element off the stack
         CL    SP,UStack          Compare to underflow address
         JNL   UndrFlow           Branch to error routine if underflow
         - - -
UStack   DC    A(Stack)           One past bottom of stack
```

Similarly, we can test for the possibility of *stack overflow* (too many elements have been pushed onto the stack).

```
         AHI   SP,-4              'Push' stack pointer one element
         CL    SP,OStack          Check against stack overflow address
         JL    OverFlow           Branch to overflow error routine
         ST    0,0(,SP)           It's safe to store on the stack
         - - -
OStack   DC    A(Stack-StkSize*L'Stack)   Overflow-test address
```

Checking for possible error conditions in manipulating stacks is strongly advised.

All these examples of stacks have used arrays as the underlying data structure. In Section 40.8, we will see how a linked list can also provide the structure needed to implement a stack.

Exercises

40.6.1.(1) Given the stack arrangement in Figure 689 on page 925, which of the following statements cannot be used to push a new element on the top of the stack, and why?

```
(1)      ST    0,Stack-4(SP)
         AHI   SP,-4

(2)      AHI   SP,-4
         ST    0,Stack(SP)

(3)      ST    0,Stack(SP)
         AHI   SP,-4
```

40.6.2.(1) Using the stack illustrated in Figure 689 on page 925 and the instruction sequence in Figure 690 on page 925, write instructions to initialize the stack and the stack pointer to the contents shown in Figure 689 on page 925.

40.6.3.(1) In Figure 685 on page 924, an element can be pushed on the top of the stack by writing code as in Figure 687 on page 924, or by writing

```
         ST    0,Stack+4(SP)
         LA    SP,4(,SP)
```

Is one method preferable to the other? Why?

40.6.4.(2)+ Show the result of evaluating the following postfix expressions, and show the contents of the operand stack at each step.

```
(1)      7 3 2 5 * * *
(2)      7 3 2 * 5 + *
(3)      7 3 + 2 5 * *
```

What would these expressions look like in “infix” notation?

40.6.5.(3)+ Suppose a postfix expression is represented in memory as an array of *pairs* of halfword integers. The first integer of the pair is a code describing the second integer: if the first integer is 0, the second is an operand; if the first is +4, the second is an operator; if the first is -1, it indicates “end-of-expression”. The values used to indicate operators are 0 for +, and +1 for *. Thus the expression 234*+ is represented by the integers

```
  0, 2, 0, 3, 0, 4, 4, 1, 4, 0, -1.
```

Assuming that this representation is stored in an array at **Expressn**, write a program segment that will evaluate the expression and leave its value in GR1.

40.6.6.(3)+ Extend your solution to Exercise 40.6.5 to include tests for valid operator codes and for the possibility of stack overflow and underflow. Also, allow for subtraction and division, which have operator codes +2 and +3 respectively.

For subtraction, treat the stack top as the minuend, the number to be subtracted from the element below it; and for division, treat the stack top as the divisor and the element below it as the dividend.

40.6.7.(3)+ Show how two stacks of word elements **Stk1** and **Stk2** can be built in a single array **A** of **N** words, in such a way that either stack could use all **N** words if the other stack is empty; the two stacks grow from each end of the array toward each other.

Show how to implement **Push1**, **Push2**, **Pop1**, and **Pop2** routines that will push and pop elements onto and from either stack, being careful to detect overflow and underflow conditions for each stack.

40.7. Lists

Most people use the word "list" to mean a linear table of items, like a shopping list. In computer-speak, a list has a more specific meaning.

The basic element of a linked list has two components, a *data* field and a *link* field. The data field may contain more than one data value. The link (or *successor* or *chain*) field contains a "pointer" to the next list element; the pointer may be its address, its offset relative to the current element, its array subscript, or whatever is appropriate to the application. Often, the actual address is used, especially when new list elements are allocated from available memory.

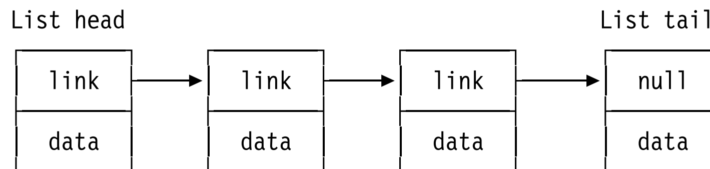


Figure 691. Sketch of a linked list

There are two ends to a list. The first element is sometimes called the *head*; it has no predecessor; and the last element, the *tail*, has no successor. The absence of a link in the link field of a list element is usually indicated by storing a special and easily identifiable *null* value such as 0 or −1.

Two basic operations on linked lists are insertion and deletion.

40.7.1. List Insertion

Suppose we have a list with elements E1 and E3 as shown in Figure 692 on page 928, and we want to insert element E2 between them.

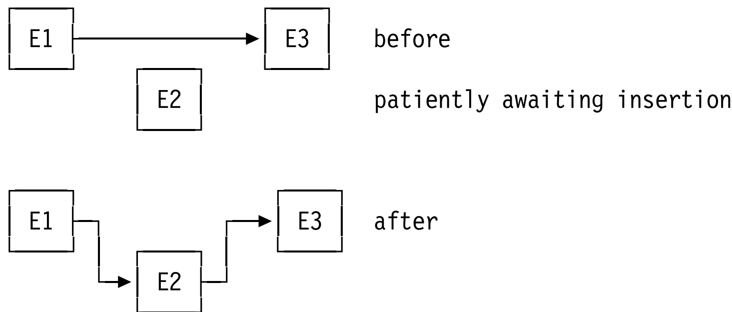


Figure 692. Inserting an element into a linked list

The insertion proceeds in two steps:

1. Move the link field pointing from E1 to E3 into the link field of E2, so that E2 now points to E3.
2. Store a link pointing to E2 into the link field of E1.

E3 is not referenced during the insertion, only the link to it from E1.

To illustrate, suppose a list element consists of two successive fullwords, the first containing the link and the second containing the data, as illustrated in Figure 691 on page 927. (The link could be the actual address of the successor element.) Suppose also that GR1 contains the address of E1, and GR2 contains the address of E2. First, we'll do this without using a DSECT:

```
Link     Equ   0                   Offset of link field in list element
         L     0,Link(,1)          Copy Link(E1) into GR0
         ST    0,Link(,2)          Link E3 to E2
         ST    2,Link(,1)          Now link E1 to E2
```

Figure 693. Example of inserting an element into a linked list

These instructions will work correctly even when E1 is the last element and E2 is to become the new tail, because the previous link field of E1 was a null link.

The example in Figure 693 uses explicit addresses in the three instructions. Now, we define a DSECT to describe the list element:

```
List_El  DSECT ,                   Describe a linked list element
Link     DS    A                   Link address to successor
Data     DS    XL20                20-byte Data field
Elem_Len Equ   *-List_El           Length of a list element
```

Figure 694. DSECT describing a list element

We will use the DSECT mapping for *each* reference to a list element. It has the benefits of DSECT mappings while letting the Labeled Using statements direct symbol resolutions to specific registers; **E1** and **E2** are symbol qualifiers.

```
E1       Using List_El,1           Map the E1 list element
E2       Using List_El,2           Map the E2 list element
         L     0,E1.Link           Load link to E3 from E1
         ST    0,E2.Link           Store link to E3 in E2's link field
         ST    2,E1.Link           Store link to E2 in E1's link field
         Drop  E1,E2               DROP both Labeled Usings
```

Figure 695. Mapping multiple list elements with Labeled USINGs

This is much simpler (and clearer) than similar statements based on Ordinary USING statements.

40.7.3. List Deletion

Removing an element from a list is very simple. To delete E2 from its position between E1 and E3, move the link field from E2 into the link field of E1, replacing the previous link to E2.

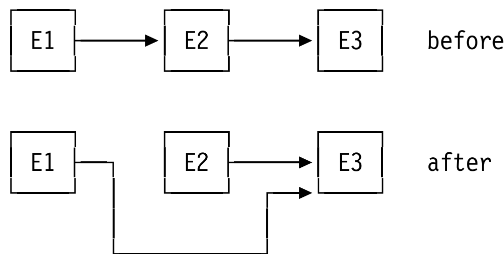


Figure 696. Deleting an element from a linked list

First, we delete E2 assuming the same register contents and definition of the **Link** offset as in Figure 693 on page 928:

```
         L     0,Link(,2)          Link from E2 to E3 in GR0
         ST    0,Link(,1)          Now link E1 directly to E3
```

Figure 697. Example of deleting an element from a linked list

These instructions work even when E2 is the last element of the list, because the link field of E2 (which then becomes the link field of E1) is then a null link.

With Labeled USINGs, the instructions in Figure 697 can be written as in Figure 698, where all symbol references are implicit.

```
E1       Using List_El,1           Map the E1 list element
E2       Using List_El,2           Map the E2 list element
         L     0,E2.Link           Load link from E2 from E3
         ST    0,E1.Link           Store link to E3 in E1's link field
```

Figure 698. Example of deleting an element from a linked list

As your data structures become more complex, Labeled USINGs and qualified symbols let you write much clearer code.

These brief examples raise three questions that we'll need to answer:

1. Where do list elements come from? For example, Figure 692 on page 928 shows a new element E2 without indicating how it was created.
2. When an element is deleted, what happens to it? In Figure 696, element E2 has no link pointing to it; is it lost?
3. How do you find the first element (the head) of the list? In Figures 692 on page 928 and 696, how should we find E1?

The simplest answers to these questions are a *free storage list* and a list *header* or *anchor*.

40.7.4. Free Storage Lists

There are may ways to obtain new list elements. Requesting the needed bytes from the program's operating environment each time you need a new element can be expensive, so the more usual technique is to create a Free Storage List, or FSL. The program first acquires memory to hold enough list elements for the application, either by defining space at assembly time or by requesting the memory space when the program starts. For simplicity, our examples will define the FSL at assembly time.

The FSL initially contains all the unused and available list elements in a single list. As elements are needed for other lists, they are removed from the FSL, and as elements are deleted from other lists, they are added back onto the FSL.

To define a free storage list of 20 two-part elements like the one in Figure 694 on page 928, we could write

```
NLstItms Equ   20                 Number of items on free storage list
FSLCount DC    A(NLstItms)        Count of free-storage list items
FSLHLink DC    A(FSLHead)         List anchor: link to head element
FSLTLink DC    A(FSLTail)         List anchor: link to tail element
*              Now, define the FSL
FSLHead  DC    (NLstItms-1)A(*+Elem_Len-4,0,0,0,0,0) All but FSL tail
FSLTail  DC    (Elem_Len/4)A(0)   Tail element with null link field
```

Figure 699. Defining a free storage list as an array

Figure 699 shows why it may be better to initialize a FSL at execution time. The statement named **FSLHead** required knowing the number of words in a list element, and the Assembler, Linker, and Program Loader must process blocks of list elements. Also, the number of elements in the FSL might be specified at execution time, when more elements can be added to the FSL as needed.

We can also initialize a FSL at execution time, as shown in Figure 700:

```
         Using List_El,1          Map the elements of the FSL
         L     1,FSLHLink         Point to first list element
         LA    0,NLstItms-1       Count (no. elements)-1 in GR0
FSLoop   LA    2,Elem_Len(,1)     Addr of next element
         ST    2,Link             Store link in this element
         LR    1,2                Move 'next' addr to 'this'
         JCT   0,FSLoop           Repeat for all but last
         ST    0,Link             Store 0 for null link in tail
```

Figure 700. Initializing a free storage list as an array

A simple way to locate the head element of a list is to maintain a separate list anchor whose location is known, and that points to the first (head) element on the list. If the list anchor contains a null link, the list it anchors is empty; otherwise it points to the head element.[293]

In addition to a pointer to the first list element, the list header might contain a count of list elements and a pointer to the last element in the list. Figure 701 illustrates such a list anchor.

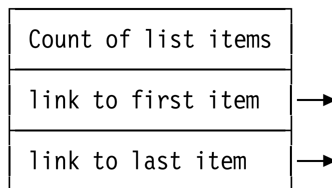


Figure 701. Example of a list anchor

The free-storage list anchor can be mapped by its own DSECT:

```
FSLDSECT DSect ,                  Map a free storage list anchor
FSLCount DC    A(0)               Count of list items
FLSHLink DC    A(0)               Link to head element
FSLTLink DC    A(0)               Link to tail element
```

Figure 702. DSECT mapping a list anchor

The “count of list items” field and the link to the list tail are optional, but they can help if you need to know how many items are in the list. Also, keeping a count of the number of free ele-

[293] Sometimes it is convenient to include the first list element in the list anchor. Then, the list is never empty, and the function of the list head is performed by the list anchor.

ments makes it easier to know when you may need to allocate more free elements. These fields can also help you check to make sure no list elements have vanished.[294] The link to the last item is useful if you need to insert an item at the end of the list, so you won't have to search the entire list to find the last element.

If your program needs frequent access to the list anchor and a general register is readily available, you can keep the anchor in a register and never need to store it.

An anchor for a "working" list could take the same form as the anchor for the FSL shown in Figure 699 on page 930:

```
WorkList DSECT ,                 Map a free storage list anchor
LstCount DC    A(0)              Count of list items
LstHLink DC    A(0)              Link to head element
LstTLink DC    A(0)              Link to tail element
```

Figure 703. Defining an anchor for a working list

As items are taken from the FSL and added to the working list, the fields of the list anchor are updated. For example, suppose we remove an element from the FSL defined in Figure 699 on page 930 and insert it at the head of the working list shown in Figure 703:

```
         - - -                   Assume c(GR9) = A(FSL anchor)
         Using FSLDsect,9        Mapping of the FSL anchor
         LT    0,FSLCount        Check count of free elements
         JNP   NoneLeft          Must take some corrective action
         L     1,FSLHLink        Addr of first free element in GR1
         Using List_El,1         Map the new list element
         L     2,Link            Link to next free element in GR2
         ST    2,FSLHLink        Previous 2nd element now FSL head
         AHI   0,-1              Decrement availability count; set CC
         ST    0,FSLCount        Store updated free-element count
         JNZ   FSLOK             FSL still has elements
         ST    2,FSLTlink        Empty FSL; update tail address also
FSLOK    DC    0H                Address of new element in GR1
         - - -                   Assume c(GR5) = A(work list anchor)
         Using WorkList,5        Mapping of the work list anchor
         L     2,LstHLink        Get link to current work list head
         ST    1,LstHLink        Link to new list-head element
         ST    2,Link            Old head becomes second element
         L     0,LstCount        Get current working list count
         AHI   0,1               Increment by 1
         ST    0,LstCount        Restore updated working list count
         CHI   0,1               Is the new element the only one?
         JNE   WorkOK            No, we're done with the work list
         ST    1,LSTTLink        Make new element be the tail also
WorkOK   DC    0H                Work list ready to use
         Drop  9,5               Release mappings of list anchors
```

Figure 704. Moving a list element from the FSL to the working list

GR1 will contain the address of the newly acquired WorkList element.

Removing an element from the working list and returning it to the FSL uses a similar sequence of instructions.

---

294 Leaving chunks of previously referenced and now inaccessible storage (known as "memory leaks") is a very poor programming practice. Some programming languages provide "garbage collection", but garbage collectors still must know what can safely be collected and what cannot.

The sum of the element counts in **FSLCount** and **LstCount** should always be the same as the number of items originally on the FSL.

Sometimes it's useful to link the list tail back to the list head, making the list *circular*. (See Exercises 40.7.2 through 40.7.6.)

If we restrict list insertion and deletion to the head element, we have an implemented a stack! This can be useful in applications where a stack must share or compete for space with other list structures, because you can allocate stack elements as needed.

The examples in Figures 699 on page 930 and 700 used addresses as links. Another way to implement a linked list is with a two-column array as the underlying structure. To illustrate, suppose we have an array of **NLItms** rows and two columns: the first column contains a "link" index to a successor element, and the second column holds the data for that element. Figure 705 shows how this might look:

Index	Link	Data
1		
2		
3		
...	:	:
NLItms		

Figure 705. A two-dimensional array to implement a linked list

We need two additional values: **HeadNdx**, the index of the first element in the working list, and **FreeNdx**, the index of the first element of the free-element list. First, we initialize the "list" so that all elements are on the "free" list:

1. Set **HeadNdx** to zero (no elements in the list).
2. Set **FreeNdx** to 1.
3. Set each Link to the index of its successor, except the last, which is set to zero to indicate the end of the free list.

The list would then look like this:

Index	Link	Data	HeadNdx = 0
1	2		← FreeNdx = 1
2	3		
3	4		
...	:	:	
NLItms	0		

Figure 706. Initializing a two-dimensional array implementing a linked list

Exercises 40.7.10 through 40.7.14 use this form of a linked list.

Exercises

40.7.1.(2) In Figure 703 on page 931 will happen to the fields a the working list anchor at **LstAnchr** if deletions remove all list elements?

40.7.2.(1) Write instructions to initialize a circular free storage list.

40.7.3.(2) Write instructions to add an element to the end of a circular list.

40.7.4.(2) Write instructions to remove an element from a circular list.

40.7.5.(2)+ Suppose you have implemented a circular list without maintaining a count field in the list anchor. How can you tell when the list is empty?

40.7.6.(2)+ Suppose you have implemented a circular list without maintaining a count field in the list anchor. How can you tell when you have processed all the elements in the list?

40.7.7.(2) Rewrite the instructions in Figure 704 on page 931 to delete the head element of the working list and return it to the FSL.

40.7.8.(3) Write an instruction sequence to take a new element from the FSL and add it to the *tail* of the working list.

40.7.9.(3) Write an instruction sequence to take an element from the tail of the working list and return it to the FSL.

40.7.10.(2) In Figure 705 on page 932, suppose the link field of each row is a word integer, and the length of the data field is **DataLen** bytes, which has been defined symbolically and is a multiple of 4. Write statements to allocate an array of **NLItms** rows (also defined symbolically), starting at **ArrStack**.

40.7.11.(3) Using the definitions in your solution to Exercise 40.7.10, write a sequence of instructions that will initialize the list at **ArrStack** as shown in Figure 706 on page 932, and initialize the values of **HeadNdx** and **FreeNdx**.

40.7.12.(3) Assume a list has been initialized as shown in Figure 706 on page 932, and that the values of **HeadNdx** and **FreeNdx** have been initialized also.

First, show the steps needed to take an element from the "free list" referenced by **FreeNdx** and assign the index of this new element to **NewNdx**, and update **FreeNdx** appropriately.

Then, write a sequence of instructions to do these steps. If no free elements are left, branch to **NoneLeft**.

40.7.13.(3) Suppose two data elements have been added to the list shown in Figure 706 on page 932, where the second element was added to the end of the list. Now, you need to add a new element with index **NewNdx** to the list, *after* the existing element in the list with index **PrevNdx**.

First, show the steps needed to put the new element into the list at the specified position. Then, write a sequence of instructions to do these steps.

40.7.14.(3) Suppose several elements have been added to the list shown in Figure 706 on page 932, and you need to remove the element with index **CurrNdx** and return it to the free storage list.

First, show the steps needed to remove the specified element from the list and return it to the free list, and updating the working list to account for the removed element.

Then, write a sequence of instructions to do these steps.

40.7.15.(2)+ Suppose the free storage list in Figure 699 on page 930 had been defined by writing

```
FSLAnchr DC    A(FSLHead)
         DC    (Elem_Len/4)A(0),(NLstItms-1)A(*-4,0,0,0,0,0)
FSLHead  Equ   *-Elem_Len
```

How would instructions using this new list differ from those using the list in Figure 699 on page 930?

40.7.16.(2)+ Write a code sequence that obtains a list element from a free storage list anchored at **FSL** and inserts it at the head of a list anchored at **ListHead**. If the free storage list is empty, branch to **AllOut**.

40.7.17.(3)+ Do as in Exercise 40.7.16, with this added requirement: the fullword integer at **DataInt** is stored in the data field of the new element, and the new element is then inserted into the list at a position where its data value exceeds none of its predecessors. That is, the list must be arranged in descending order of data items.

40.8. Queues

Linked lists occur in many forms; an important form uses double chaining. We saw in the previous examples that simple lists can be scanned only in the "forward" direction, starting at the head and chaining toward the tail.

It is often useful to be able to move in both directions, so the list elements must contain both a forward link and a backward link. We call this doubly-chained structure a *queue*; elements can be added and removed at both ends of such a list.

Figure 707 illustrates the structure of a queue element:

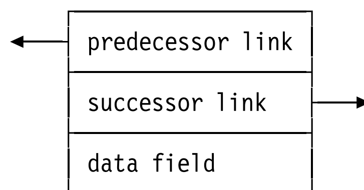


Figure 707. Structure of a queue element

A queue element contains links to its predecessor and successor elements (sometimes called "forward" and "backward" links), as sketched in Figure 708.

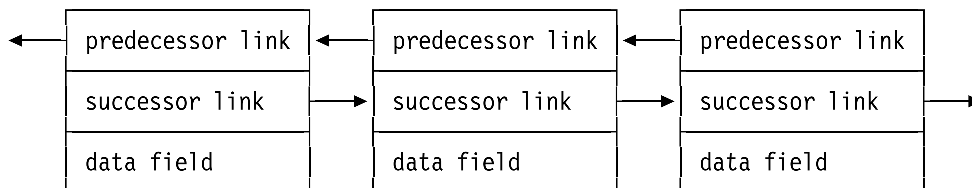


Figure 708. A queue with several elements

Adding and deleting list elements other than the head or tail is more complicated than for singly-linked lists, because four links involving three elements must be manipulated. For example, suppose we have defined a DSECT describing the queue elements with these statements:

```
Q_El     DSECT ,                 Mapping of a queue element
LLink    DS    A                 Link to (left) predecessor element
RLink    DS    A                 Link to (right) successor element
ElemData DS    XL(DataLen)       Space for the element's data
```

Figure 709. DSECT structure of a typical queue element

Now, suppose also that we have located two elements of the queue pointed to by GR2 and GR3, and we want to insert the new element pointed to by GR5 between them, as in Figure 710 on page 935.

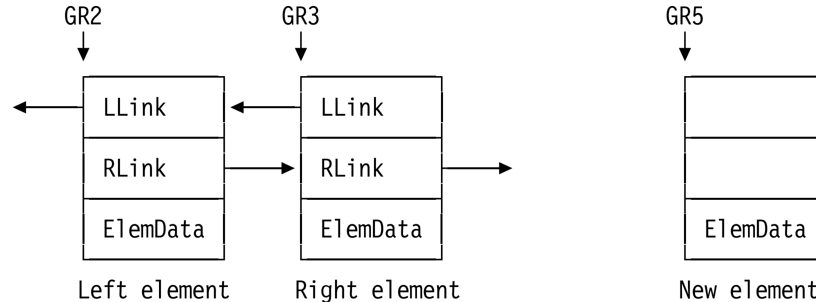


Figure 710. An element to be inserted into a queue

After the new element has been inserted, we want the result to look like this, where the links that must be changed during the insertion process are indicated in Figure 711 by the numbered keys **1** and **2** (the "left" links), and by **3** and **4** (the "right" links).

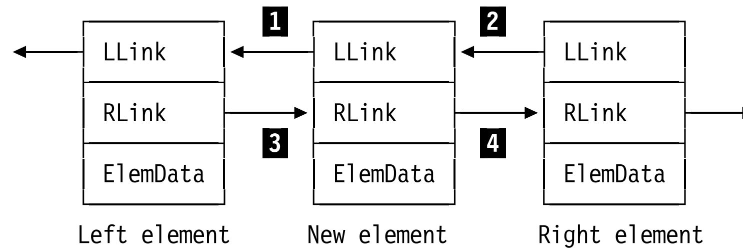


Figure 711. A queue after insertion of a new element

Four steps are needed to do the insertion:

1. Copy the **LLink** (predecessor link) field from the Right element into the **LLink** field of the New element (**1**). The predecessor of New is now Left.
2. Store a **LLink** to the New element into the **LLink** field of the Right element (**2**). The predecessor of Right is now New.
3. Copy the **RLink** (successor link) field from the Left element into the **RLink** field of the New element (**4**). The successor of New is now Right.
4. Store a **RLink** to the New element into the **RLink** field of the Left element (**3**). The successor of Left is now New.

If we use actual addresses as the links, we could do this (where the numbered steps above are shown in the comments fields):

```
L     0,LLink-Q_El(,3)      Get left link from Right element
ST    0,LLink-Q_El(,5)   1  Store as left link of New element
ST    5,LLink-Q_El(,3)   2  Store left link from Right to New
L     0,RLink-Q_El(,2)      Get right link from Left element
ST    0,RLink-Q_El(,5)   4  Store as right link of New element
ST    5,RLink-Q_El(,2)   3  Store right link from Left to New
```

Figure 712. Instructions to insert a new queue element

This style of coding is awkward and error-prone because explicit addressing is used for every instruction. If any of the registers must be reassigned, every instruction using that register must be found and updated.

We could use ordinary USING statements to refer to the fields in each element with proper symbolic addressing:

```
Using  Q_El,3          Map RIGHT element
L      0,LLink         Save  old Right.LLink
ST     5,LLink         Store new Right.LLink  2
Drop   3               Unmap RIGHT element
Using  Q_El,2          Map LEFT element
L      1,RLink         Save  old Left.RLink
ST     5,RLink         Store new Left.RLink   3
Drop   2               Unmap LEFT element
USING  Q_El,5          Map NEW element
ST     0,LLink         Store new New.LLink    1
ST     1,RLink         Store new New.RLink    4
Drop   5               Unmap NEW element
```

Figure 713. Insert a new list element with ordinary USINGs

The primary shortcomings of this method are

- intermediate temporaries (in this case, registers 0 and 1) are used to hold some of the pointers;
- it requires a precise sequence of USING and DROP statements to obtain correct address resolutions;
- two additional instructions are required (Load and Store via registers).

To eliminate the need for intermediate temporaries, we can use MVC instructions to move the fields (with a presumed gain in efficiency):

```
 - - -                        R5 points to New element
 USING  Q_El,5                Map NEW element
 MVC    LLink,LLink-Q_El(,3)  Move  old Right.LLink  1
 ST     5,LLink-Q_El(,3)      Store new Right.LLink  2
 MVC    RLink,RLink-Q_El(,2)  Move  old  Left.RLink  3
 ST     5,RLink-Q_El(,2)      Store new  Left.RLink  4
```

Figure 714. Ordinary-USING Code to Insert a New List Element

This sequence contains the "efficient" instructions, but its defects are:

- greater difficulty of understanding;
- increased likelihood of maintenance problems due to fixed register assignments in the instructions.

The simplest and clearest solution is to use Labeled USINGs, with appropriate descriptive qualifiers **Left**, **Right**, and **New**. Thus, three instances of the DSECT named **Q_El** are concurrently active.

```
         - - -                      R5 points to New element
Left     Using Q_El,2               Labeled USING for Left element
Right    Using Q_El,3               Labeled USING for Right element
New      Using Q_El,5               Labeled USING for New element
         - - -
         MVC   New.LLink,Right.LLink   1    Link from New to Left
         ST    5,Right.LLink           2    Link from Right to New
         MVC   New.RLink,Left.RLink    3    Link from Left to New
         ST    5,Left.RLink            4    Link from New to Right
```

Figure 715. Labeled USING example: inserting a new queue element

The advantages in clarity, readability, simplicity, and improved ease of maintenance are clear. The only references to specific registers are in the USING statements. Without Labeled USINGs, the code for these operations is more convoluted, and difficult to read, understand, and maintain.

A free storage list for queue elements can be the same as for a singly-linked list, because the free-list elements only need to be chained in one direction.

Queues are sometimes "circular", where the last element is linked to the first and the first to the last. This allows searching in either direction, starting at the queue head.

Exercises

40.8.1.(2)+ Describe the steps necessary to delete element Q2 between elements Q1 and Q3 in a queue. Use the queue-element definition in Figure 709 on page 934.

40.8.2.(3)+ Suppose you have defined a queue of elements with a 4-byte data field containing a word integer. The elements are ordered along the forward links in descending value: that is, the data in the element pointed to by the forward link (if it exists) is less than the value in the element containing the link. The list anchor points to an arbitrary element of the queue. Write instructions that search the queue for an element whose data item matches the word integer at **Value**. If no match exists, branch to **InsertIt**.

40.8.3.(4) When inserting elements into an ordered queue, the list may become *unbalanced* if most of the entries are added on one side of the "center" element pointed to by the anchor. To remedy this, maintain a flag indicating which side of the "center" received the most recent insertion. If two successive insertions occur on the same side, the anchor is changed so that the "center" element is moved one element in that direction.[295]

Write an instruction sequence that constructs such a balanced queue from the array of 100 fullword integers starting at **DataVals**.

40.9. Trees

Trees are structures whose elements may contain pointers to more than one "successor" element. In a *binary tree,* for example, each element (or *node*) contains a data field and links to at most two successors. Because the term "successor" is not well defined when more than one is possible, we sometimes use a more "personified" terminology and call the successors of the "parent" node its "left" and "right" children. One parent node is distinguished by having no parent of its own; it is called the *root* of the tree.

The literature on trees is vast, so we will simply give a few examples to illustrate possible implementations. The format of a typical tree element is illustrated in Figure 716.

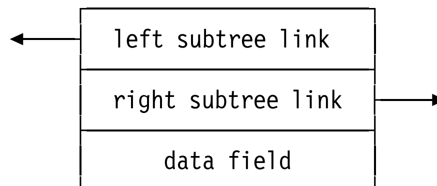


Figure 716. Node of a binary tree

We can map the structure in Figure 716 with a DSECT like that in Figure 709 on page 934:

```
Tree_El  DSECT ,                 Mapping of a tree element
LLink    DS    A                 Link to left subtree
RLink    DS    A                 Link to right subtree
ElemData DS    XL(DataLen)       Space for the element's data
```

Figure 717. DSECT structure of a typical tree element

The topmost node of a binary tree is its "root" node. All accesses to nodes in the tree start at the root. An example of a binary tree with 3 nodes is shown in Figure 718 on page 938.

[295] This technique was used in the old IBM Fortran H compiler to construct its symbol tables.

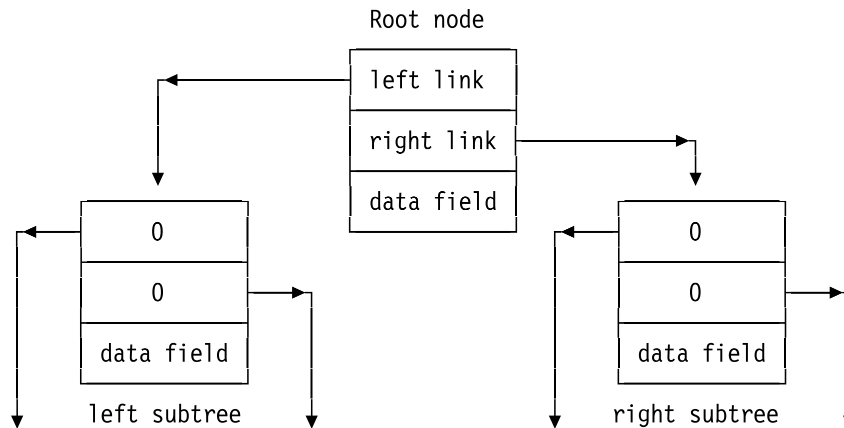


Figure 718. Three nodes of a binary tree

This might look like a three-node queue with “empty” links, but a tree with more nodes might look like Figure 719; we see that a tree structure can't be represented by a queue.

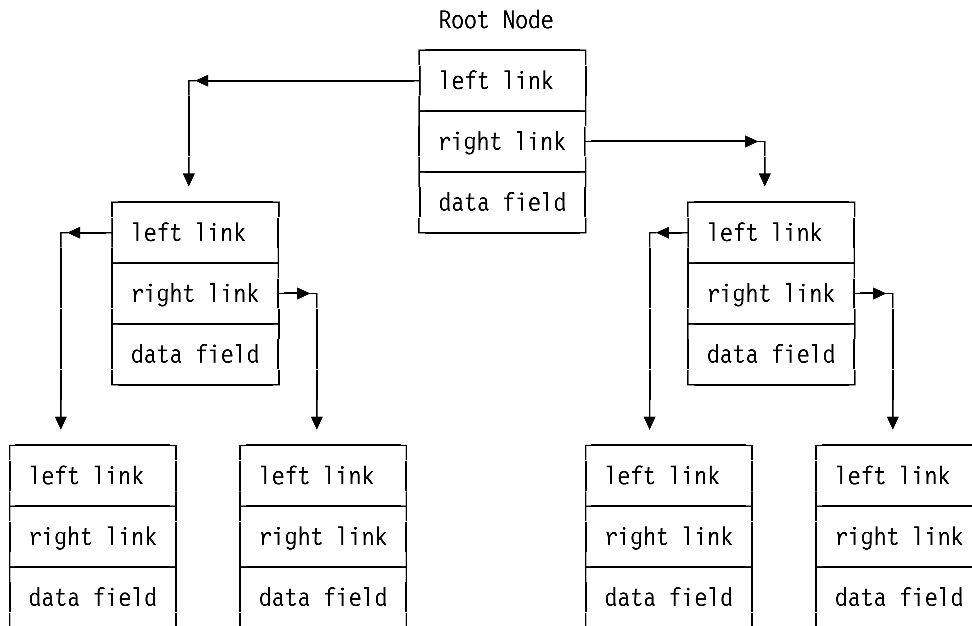


Figure 719. A growing binary tree with seven nodes

Suppose we want to construct a symbol table using a binary tree, so that the symbols are maintained in alphabetic order. If we look at the symbol in a given node, all the symbols in the “left subtree” (that is, the left subtree and its subtrees) precede the given symbol in alphabetic order, and all the symbols in the “right subtree” and its subtrees follow the given symbol. Assume that the tree elements are composed of three successive words, as illustrated in Figure 716 on page 937. Let GR0 contain a 4-character symbol to be entered into the tree, and suppose GR1 contains the address of an empty node obtained from a free storage list. As additional simplifications, we suppose the tree already contains some entries, and if one of them matches the new symbol, we will branch to **FoundIt**.

```
New      Using Tree_El,1         Map the new node based on GR1
RW       Equ   3                 Use GR3 as a work register
         L     RW,RootPtr        Pick up pointer to root node
Compare  LR    2,RW              Point to node to be compared
         Using Tree_El,2         Anchor the tree element mapping
         CL    0,ElemData        Compare symbol to current node
         JL    GoLeft            Symbol smaller, search left subtree
         JE    FoundIt           Symbol already in tree, exit.
GoRight  L     RW,RLink          Get link to right subtree
         LTR   RW,RW             Check for right subtree
         JNZ   Compare           Right exists, scan right subtree
         ST    1,RLink           New node becomes right subtree
         J     Store             And go store symbol
GoLeft   L     RW,LLink          Get link to left subtree
         LTR   RW,RW             Check whether it's there
         JNZ   Compare           Branch if yes to search
         ST    1,LLink           New node becomes left subtree
Store    ST    0,New.ElemData    Store symbol into new node
         XR    0,0               Clear GR0 for storing new links
         ST    0,New.LLink       Clear left and right links of...
         ST    0,New.RLink       ...new node: it has no subtrees
         - - -
FoundIt  - - -                   Data already entered in the tree
```

Figure 720. Entering a new node in a binary tree

This method can be inefficient: if the symbols being entered in the tree are received in ascending alphabetic order, we will generate a tree that is actually a linear list, chained along the right-tree links. Techniques for balancing binary trees are beyond the scope of this section, but are described in many programming references and textbooks if you're interested.

Having constructed a tree, we will need to search it to retrieve the symbols in alphabetic order. Because all left subtrees precede their parents, and the right subtrees follow their parents, we must scan the left subtrees of a node before “processing” the data in that node, followed by a scan of the right subtrees. There are two problems to be solved: how to “travel” through a tree structure, and how to keep track of which cildren of a node have been visited, and which ones are still to be processed.

As we work our way from the root toward the lower levels of the tree, we must retain some information on how to return to the parents of the nodes already passed. This problem is easily solved if each node contains a link to its parent, but this requires an extra link field in each node. Similarly, if some space outside the data structure is available, we can also maintain a stack of return links; each time we move from a parent node to one of its children, the address of the parent node is pushed onto the stack, to be popped off when we return to the parent. A third technique is to reverse the links as we pass over them: when moving from parent to child, one of the links in the child node is saved in a register, and the link field is replaced by a pointer to the parent node. Then, when returning from child to parent, the links are restored to their original and correct orientation. (This method cannot be used if the tree may be traversed by more than one process at a time, as might occur in large application programs such as reservation systems and similar data bases.)

The second problem, keeping track of which subtrees of a node are still to be processed, requires maintaining some kind of counter at each level. If we are using a stack to hold return links, we can push the number or address of the next subtree to be processed onto the stack at the same time. Otherwise, a field in the node must be available to hold the count. In a binary tree the counter can be a single bit.

We will traverse the binary tree constructed in Figure 720, extract the symbols in alphabetic order, and store them in a table of fullwords starting at **Ordered**. Assume there is a sufficiently large array of fullwords at **LStack** that can be used to hold the return links. Since the tree is binary, we need not maintain a count; to indicate that a node has been “processed”, the return link on the stack will be stored as a negative value.

```
RW       Equu  3                  Use GR3 as a work register
RTbl     Equ   1                  Pointer to output table
RStk     Equ   4                  Points to word past top of stack
         L     RW,RootPtr         Initialize work pointer to root
         LA    RStk,LStack        Initialize link stack pointer
         LA    RTbl,Ordered       Initialize output pointer
         SR    2,2                Node pointer = 0 indicates root
StackLnk ST    2,0(,RStk)         Put current node pointer on stack
         LA    RStk,4(,RStk)      Bump stack pointer by 4
         LR    2,RW               Move to left subtree node
         Using Tree_El,2          Map the tree node's structure
LookLft  L     RW,LLink           Get its left subtree link address
         LTR   RW,RW              Does its left subtree exist...
         JNZ   StackLnk           If yes, stack link and continue
Output   L     0,Data(,2)         'Process' the data from the node...
         ST    0,0(,RTbl)         ...by storing it into the table
         LA    RTbl,4(,RTbl)      Increment table storage address
LookRgt  L     RW,RLink           Get the right subtree link address
         LTR   RW,RW              Does the right subtree exist...
         JZ    Return             Branch if not to return to parent
         LCR   2,2                Form negative value for stack
         J     StackLnk           Go stack link, look leftward
Return   AHI   RStk,-4            Pop stack pointer once
         L     2,0(,RStk)         Retrieve popped return link
         LTR   2,2                Check for already processed
         JM    Return             Branch if yes, pop again
         JP    Output             Go process the node if available
         - - -                    Otherwise if zero, we're all done
LStack   DS    20F                Stack for node links
Ordered  DS    100CL4             Ordered output values
```

Figure 721. Retrieving data from a binary tree

The examples in Figures 720 on page 939 and 721 have used one of three common ways to traverse a binary tree. We assumed that each left subtree contains elements less than that of the parent node, and each right subtree contains elements greater than the parent node's. This is called "inorder" traversal. The other two forms are called "preorder" and "postorder". The order of traversal for each of the three forms is

- preorder: parent, left subtree, right subtree
- inorder: left subtree, parent, right subtree
- postorder: left subtree, right subtree, parent

Consider the small tree with 7 elements shown in Figure 722, in which three elements have "children" and four do not (these are sometimes called "leaf nodes").

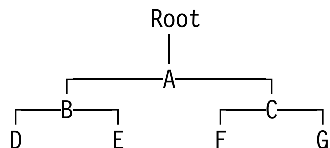
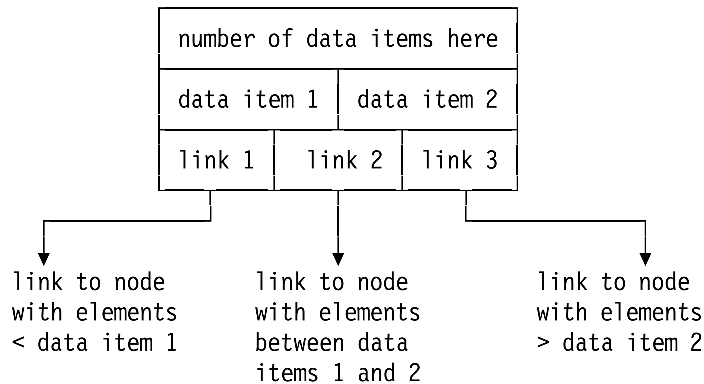


Figure 722. Example of a binary tree of 7 elements

The order in which the elements are visited is

- preorder traversal: A B D E C F G
- inorder traversal: D B E A F C G
- postorder traversal: D E B F G C A

Other types of trees with more than one data element and more than two links per node are widely used; they are sometimes called "B-trees" or "N-trees". For example, a B-tree node with two data elements and three links might look like this:



You will need to know the number of data items and links in a node so you can know how many comparisons are needed, either to locate an item in this node or to follow a link to a lower node.

B-trees can mean that fewer nodes need to be searched to locate a particular data item.

Exercises

40.9.1.(2) The description of tree traversal methods always visits a "left" subtree before a "right" subtree. What would happen if the order of left and right were interchanged?

40.9.2.(2) Figure 718 on page 938 shows a binary tree with 3 nodes. How many other 3-node binary trees are possible?

40.9.3.(2) Now that you have solved Exercise 40.9.2, suppose your 3-node binary trees contain the symbols A, B, and C in alphabetic order. If the trees are traversed using "inorder" traversal, sketch what your trees will look like.

40.9.4.(3) Suppose an element of a tree can have up to four subtrees, of which the values in the first two precede it and the values in the last two follow it, in whatever ordering is applied to the contents of the data field. In addition, the subtrees are ordered (by the same relation) from "left" to "right". What additional fields are required in the node? What are the advantages of such a representation over a binary tree?

40.10. Hash Tables

As with the other structures, the literature on hash tables is large. We will give only enough background here to illustrate some implementation techniques for System z.

All of the previously described data structures suffer from one important defect: to locate an element, the structure must be searched in an orderly and sometimes sequential fashion. We have seen how to search arrays and lists linearly, and how to use binary trees, or binary search in an array, to search more efficiently. When the number of data elements is large, however, even these techniques become less efficient.

If the data need not be stored and retrieved in an ordered way, and if few deletions are performed, *hash addressing* or *hashing* is an attractive alternative. This "associative" form of addressing locates the element directly by its contents rather than by its position in an array, list, or tree.

There are many "hash functions". The one you use is unimportant, so long as it creates a reasonably uniform distribution of hash values: you don't want too many data items to have the same hash value, because your hash table will have too many items in some areas and too few in others.

The item to be inserted or found in the table (called the *key*) is used to form a "numeric hash", a pseudo-random number generated from the datum itself. For example, the bits of the key may be hashed by multiplying or dividing, or by adding or XORing pieces of the key together. This *hash*

value is then used as an index to look into (or probe) a table. If the probed position is vacant, the datum is entered there; if it is not vacant and the datum occupying that position does not match the key, a *collision* has occurred. We can make further probes to find either a match or an empty place in the table, or create another data structure anchored at the hash position.

Figure 723 gives an example of a hash function; another is shown in Figure 724 on page 943. Suppose the word at **DataItem** should be placed into a hash table with 73 entries. Our hash function will XOR the two halves of the data item, and divide the result by the number of table entries to create the hash-table index.

```
HashEnts Equ   73                 Number of hash table entries
         XR    0,0                Set GR0 to 0
         LR    1,0                And also GR1
         ICM   0,3,DataItem       First 2 bytes of data item in GR0
         ICM   1,3,DataItem+2     Last 2 bytes of data item in GR1
         XR    1,0                XOR the halves into GR1
         XR    0,0                Clear GR0 for a division
         D     0,=A(HashEnts)     Divide by hash table size
*                                 Hash index is now in GR0
```

Figure 723. Example of searching a hash table

Suppose we have two fullword arrays starting at **Symbols** and **Count**, containing **NE** entries each. Each element in the **Symbols** array may contain a 4-character alphanumeric symbol, and each element of the **Count** array is an integer count of the number of occurrences of that symbol. We wish to test if the symbol stored at **InData** appears in the table: if not, it will be entered and its count will be set to 1; if there already, its count will be incremented by 1. A few further points remain:

1. The number **NE** of table entries is usually chosen to be prime, giving a more uniform distribution of entries. We will use as a hash value the remainder from dividing the EBCDIC representation of the key by **NE**.
2. If we encounter a non-matching table entry (a collision), we will probe further simply by moving to the next higher position in the table. If we run off the top, we begin again at the bottom.
3. If no vacant position remains, and no match occurs after searching the entire table, branch to **FullUp**.

In Figure 724 on page 943, GR6 contains the hashed value of the key symbol; GR2 contains the symbol being sought; GR0 contains zero (to test for vacant positions); and GR1 is used as an index for the currently probed position of the table.

```
NE       Equ   101               (Prime) Number of possible entries
         SR    0,0               Initialize to zero
         L     2,InData          Load search key
         LR    6,2               Prepare to calculate hash value
         SRDL  6,32              Shift logically, GR6 zeroed
         D     6,=A(NE)          Divide by table size
         SLL   6,2               Form fullword index
         LR    1,6               Initialize search index
Probe    CL    0,Syms(1)         See if empty slot
         JE    NewSym            Branch if yes, store symbol
         CL    2,Syms(1)         Is symbol already there?
         JE    Match             Branch if in table
         LA    1,L'Syms(,1)      Move index to next position
         CL    1,=A(NE*L'Syms)   Compare to top of table
         JL    NotOver           Branch if not over
         SR    1,1               Reset index to bottom if over
NotOver  CLR   1,6               Check for no empty slots at all
         JNE   Probe             Loop if not finished
         J     FullUp            No space left
NewSym   ST    2,Syms(1)         Store new symbol in table
Match    L     2,Count(1)        Get count field
         AHI   2,1               Increment by 1
         ST    2,Count(1)        And restore the counter
         - - -
Syms     DS    (NE)F             Define table for symbols
Count    DS    (NE)F             And for counts
```

Figure 724. Example of searching a hash table

This example isn't very useful, because the hash table cannot hold more data items than the number of elements.

Hashing can be advantageously combined with other structures. For example, if each element of the **Syms** array is the anchor for a linked list or a binary tree, the table can hold many more items without much increase in search time. In such cases, we needn't worry about secondary probes in the event of a collision; the list or tree anchored at that position is searched for a match, and if none exists a new element is added to the structure.[296] Searching such structures can be very efficient.

As with the other structures, the literature on hash tables is large.

Exercises

40.10.1.(2) Reduce the number of instructions in Figure 723 on page 942 by using shifts.

40.10.2.(2)+ In Figure 723 on page 942, what will happen if the two ICM instructions are replaced by LH instructions? Will a valid hash index be generated?

[296] This method, with a linked list anchored at each hash table entry, was used to construct the symbol table in the original System/360 IBM “F-level Assembler”.

40.11. Summary

The variety of data structures is huge; this section has tried to show how to handle some of the more common types.

Terms and Definitions

address table
A table of addresses of individual rows or columns of an array, allowing faster access to the elements of that row or column.

array
A collection of data items usually of the same data type and length, arranged in contiguous storage locations. Usually accessed using one or more index variables or “subscripts”.

B-tree
A tree whose nodes contain more than one data element, and more than two links to successor nodes.

binary search
A technique for searching ordered arrays by probing the midpoint of successively smaller portions of the array.

binary tree
A data structure in which each element contains links to two other elements, a “left subtree” or “left child”, and a “right subtree” or “right child”.

column order
A way to store arrays so that the elements of each column follow one another in memory. The normal ordering for one-dimensional arrays. For arrays of two or more dimensions, subscripts cycle most rapidly from left to right.

column-major order
Same as *column order*.

double-ended queue
Same as *queue*. Sometimes called a “dequeue”.

doubly-linked list
Same as *queue*. Sometimes called a “double-threaded” list.

free storage list
A list containing unused and available elements. Abbreviated *FSL*.

hash function
A function that creates a randomized linear subscript from a data element. Used to avoid lengthy table searches for large or complex data items.

hash table
A table of data items (possibly including lists and pointers to other data items or structures) whose entries are accessed using the results of a *hash function*.

infix notation
The traditional form of writing arithmetic expressions, where operators are placed between operands, as in `2*(3+4)`.

inorder tree traversal
A technique for traversing a binary tree, visiting first the left subtree, then the parent node, and then the right subtree.

linear subscript
For arrays of two or more dimensions, the evaluation of a subscript that treats the array as having been mapped into a one-dimensional array corresponding to the CPU's linear arrangement of bytes in memory.

linked list
Same as *list*. Sometimes called a “single-threaded” list.

list
A sequence of data elements each containing a link to its successor. If the first and last elements are identified and the next element to be accessed is the last, sometimes called a "First In, First Out" (FIFO) list.

postfix notation
A representation of expressions convenient for evaluation. The infix form `2*(3+4)` is represented as `2 3 4 + *`.

postorder tree traversal
A technique for traversing a binary tree, visiting first the left subtree, then the right subtree, and then the parent node.

preorder tree traversal
A technique for traversing a binary tree, visiting first the parent node, then the left subtree, and then the right subtree.

queue
A sequence of data elements each containing links to its successor and to its predecessor. Sometimes called a "doubly-linked list."

row order
A way to store arrays so that the elements of each row follow one another in memory. For arrays of two or more dimensions, subscripts cycle most rapidly from right to left.

row-major order
Same as *row order*.

stack
A data structure with a single visible element, the "stack top". Sometimes called a "Last In, First Out" (LIFO) list or queue.

table
A term often used to describe a one-dimensional array whose columns may contain a mixture of different data types and lengths.

virtual origin
The address of a (possibly nonexistent) array element all of whose subscripts are zero.

Programming Problems

Problem 40.1.(3) This problem requires that you plot a graph to occupy a full page of printed output that we'll assume is a two-dimensional array 60 lines high and 120 characters wide (that is, 60 rows and 120 columns). Suppose we divide the page into 119 horizontal divisions (columns) corresponding to X values in the range $-59 \leq X \leq +59$, and 59 vertical divisions (rows) corresponding to Y values in the range $-29 \leq Y \leq +29$.

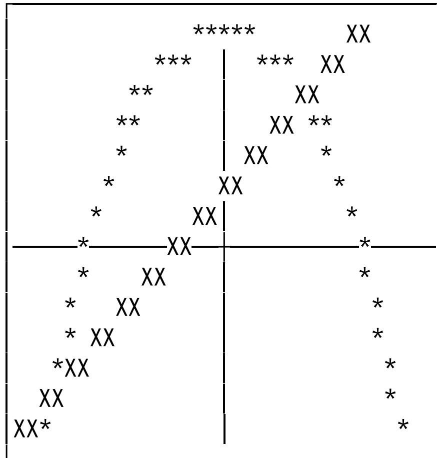
Set the first character in each line to a blank character, except on the first line, set it to `C'1'`.

On this "page" you should plot

1. an X-axis (use minus signs)
2. a Y-axis (use capital letter I's or vertical bars)
3. the Y-values corresponding to the functions

```
(a) Y = (X/2) + 6 (use X's for the points), and
(b) Y = 25 - (X*X)/50 (use asterisks for the points).
```

If a Y-value lies off the page, plot nothing. Your graph will look roughly like an inverted parabola with a diagonal line through it, as in the following sketch (so you can tell whether your graph is upside down or otherwise lopsided):



Use the PRINTLIN macro instruction to print the 59 lines.

Problem 40.2.(3) Write a program to evaluate *logical* expressions of the form used in symbolic logic. For example, if "&" represents AND and "|" represents OR, and A, B, and C are logical variables, we might ask if the expression

```
A&B&(C|A)            Infix notation
```

is true for all possible true-false values of the three variables.

Let the possible values of variables be 0 (meaning FALSE) and 1 (meaning TRUE). We can evaluate logical expressions in operator postfix notation using a stack; the above expression would then be written

```
AB&CA|&              Postfix notation
```

Use the values `B'11110000'`, `B'11001100'`, and `B'10101010'` to represent A, B, and C respectively, and let `B'11111111'` and `B'00000000'` represent the constants 1 (TRUE) and 0 (FALSE) respectively.

Your program should read data records containing logical expressions in postfix notation, and evaluate them using a byte stack (not a word stack). If the final value is `B'11111111'` the expression is always true, and if the value is `B'00000000'` the value is always false; otherwise it is indeterminate. Print the original postfix expression and a message describing the result of the evaluation. Try your program with the expressions `ABC01||||` and `ABC01&&&&` and as many others as you can devise.

Some possible extensions:

1. Support the unary operator N (NOT) that forms the one's complement of the element on top of the stack, and the binary operator X (XOR).
2. Create a stack of word entries, and define additional variables D and E appropriately.

Problem 40.3.(3) Write a program that will read 4-character symbols from the first 72 columns of a record (there may be at most 18 symbols to a record). A "symbol" of all blank characters indicates the end of the record. Insert the symbols in *alphabetical order* into a linked list whose elements have two data fields (4 bytes for the symbol, and a word binary integer count) and a fullword link field.

As the symbols are read from the record, scan the list. If the symbol is in the list, increment its count by 1. If the symbol is not in the list, obtain a free element from a FSL, place the symbol in it, initialize its count to 1, and insert it in the list in the correct position. If a symbol on the record starts with an asterisk, print the contents of the list, giving each symbol and its count.

After the data in an element has been printed, return the element to the free storage list. Thus, when the printing is done, all the data elements will be back on the FSL, and the data list will be empty. Then read some more data records and build a new symbol table, until no more data records are available.

Problem 40.4.(3) Write a program like that of Problem 40.3, but use a hash table to store the symbols. You will have to devise a method to sort the symbols into alphabetic order before printing them.

Problem 40.5.(4) Write a program like that of Problem 40.3, but use a binary tree to store the symbols.

Problem 40.6.(3) A matrix is a two-dimensional array, as described in "40.2. Two-Dimensional Arrays" on page 910. Matrix multiplication is a common problem in data analysis; to calculate the product matrix C of two N-by-N matrices A and B, we use the following formula:

```
C(i,j) = Σ (k=1,N) A(i,k) * B(k,j)
```

Write a program to evaluate a product matrix containing fullword integer values. For example, let N be 5, and initialize the A matrix with rows containing 1, 2, 3, 4, and 5, and initialize the B matrix with rows containing 5, 4, 3, 2, and 1. Then test your program with A and B with all elements initialized to 2. Assume that all sums and products don't exceed 30 significant bits.

Problem 40.7.(4) Do the same as in Problem 40.6, but this time move as much of the subscripting arithmetic as possible outside the loops. Then, use Branch on Index instructions to increment and test the loop indices. Compare this solution to your solution to Problem 40.6.

Problem 40.8.(3) Write a program to print a square centered on a 120-character print line (with one additional initial character for carriage control spacing), centered on a 60-line page. The square is 12 by 12; and its outer border is 12 `'1'` characters, and its inner border is 10 `'0'` characters. The rest of the page is blank. (Ignore the fact that character spacing on the printed page may be different in horizontal and vertical directions.) For example, the upper left corner of the square would look like this:

```
11111
10000
10
10
10
```

For extra credit: determine the actual character and line spacings used on your printer, and adjust your printed output to more accurately resemble a square rather than a rectangle.

Problem 40.9.(3) Do as in Problem 40.8, but rotate the "square" by 45 degrees to create a diamond-shaped rhombus. Make the outer border 18 by 18 characters. The top of the diamond would look like this:

```
   1
  101
 10 01
10   01
```

For extra credit: determine the actual character and line spacings used on your printer, and adjust your printed output to more accurately resemble a rotated square rather than a rhombus.

Problem 40.10.(3) Write a program to print a circular ring centered on a 120-character print line (with one additional initial character for carriage control spacing), and centered on a 60-line page. The outer diameter of the ring is 60 characters (so it will fill a 60-line page). The width of the ring should be 10 characters, so there will be an empty inner circle with diameter 40 characters. Use `'*'` characters to fill the ring, and leave the rest of the page blank.

For extra credit: determine the actual character and line spacings used on your printer, and adjust your printed output to more accurately resemble a true circle rather than an ellipse.

Problem 40.11.(3)+ This problem uses the representations introduced in Exercise 40.6.4. Write a program that reads expressions in postfix form from data records, evaluates the expression, and prints the result. The data records are prepared according to these rules: (1) all operands are positive integers of 5 or fewer digits; (2) all operators and operands are separated by a single blank; (3) the operators `+ - * /` are represented by themselves; (4) the data record ends in

column 72. Your program should check for underflow and overflow; include some data to verify that errors are correctly detected.

Problem 40.12.(2) Build a table of the first 25 members of an integer sequence defined by `S(N)=S(N-1)+S(N-2)+S(N-3)`, where `S(1)=0`, `S(2)=1`, and `S(3)=2`. Then, format and print all 25 members of the sequence; the largest is 1166220.

Then, "dump" the table and check that you can verify the values in hexadecimal.

Problem 40.13.(3)+ Revise your solution to Problem 40.12, but this time put the address of the table in a register and *don't change it*: all references to the table must be made using only its address (as though the table is somewhere unknown). After building the table, format and print the members of the sequence. Eliminate leading zeros from the results.

Problem 40.14.(3) Do as in Problem 40.13, but now build a second table containing the formatted values as character strings. Then, print the character values by stepping through the array of character values.

Chapter XII: System Services, Reenterability, and Recursion

```
XX       XX  IIIIIIIIII  IIIIIIIIII
XX       XX  IIIIIIIIII  IIIIIIIIII
 XX     XX       II          II
  XX   XX        II          II
   XX XX         II          II
    XXXX         II          II
    XXXX         II          II
   XX XX         II          II
  XX   XX        II          II
 XX     XX       II          II
XX       XX  IIIIIIIIII  IIIIIIIIII
XX       XX  IIIIIIIIII  IIIIIIIIII
```

The two sections of this chapter discuss some advanced topics:

- Section 41 sketches several types of operating system functions and how your programs can utilize them.
  - General characteristics of macro instructions used to request Operating System services.
  - Purposely causing abnormal program termination.
  - Simple forms of storage management: how to acquire and release blocks of memory.
  - An overview of key elements of input and output.
  - Techniques for managing program interruptions, and a description of handling abnormal terminations of any kind.
- Section 42 describes:
  - Program reenterability: what it means and why it can be important.
  - Program recursion, and how to write programs to handle it.

41. Using System Services

```
     444          11
    4444         111
   44 44        1111
  44  44          11
 44   44          11
44444444444       11
444444444444      11
      44          11
      44          11
      44          11
      44      1111111111
      44      1111111111
```

This section provides a brief overview of these topics:

- Macro instructions used to access system services, and their typical formats
- The SVC and PC instructions
- Voluntary and involuntary abnormal termination of your program
- Some basics of storage management
- A short introduction to one common form of sequential Input/Output
- Ways to handle some exceptions

This is only a sample of the many, many system services available to you. For all system services, you should have the relevant manuals available.

41.1. Invoking System Services

These two instructions are most often used to invoke operating system services:

Op	Mnem	Type	Instruction	Op	Mnem	Type	Instruction
0A	SVC	I	Supervisor Call	B218	PC	S	Program Call

Table 431. Supervisor and Program Call instructions

The SVC instruction has the form shown in Table 432. It invokes the operating system Supervisor by causing a Supervisor-Call interruption.

0A	I

Table 432. SVC instruction

It is written with only a single operand, in the form

```
         SVC   I
```

where the second byte **I** of the SVC instruction contains an 8-bit number that is placed in the Interruption Code portion of the “old PSW” in the fixed area at the low-address end of main memory. (You may recall from Section 4.5 that Supervisor Call is one of the six classes of interruption: that is, execution of an SVC instruction invariably causes that type of interruption.) The new PSW gives control first to an “Interrupt Handler” that saves the registers and the old PSW in

a safe place, and then passes control to a routine that examines the 8-bit Interruption Code and decides what function is desired.

The Program Call instruction doesn't cause a program interruption, so it can sometimes operate more efficiently than SVC. It normally causes a change from problem to supervisor state. Its format is shown in Table 433:

B218	B_2	D_2

Table 433. Program Call instruction

Its single operand is

```
PC    D2(B2)
```

so it can handle at least 2^{24} possible operand values. Conversely, the 8-bit I field of SVC can support at most 256 possible values, some of which are reserved for customer use.[297]

Usually, further information such as flags and values, or addresses of other data, is placed in the general registers just prior to executing the SVC or PC; registers 0 and 1, and sometimes 14 and 15, are almost always used. In some cases the Supervisor will place values into one or more registers before returning control to the program that executed the SVC or PC.

41.2. Invoking System Services with Macro Instructions

Most operating system services are invoked using *macro instructions*, or “macros”.

A macro instruction is an invocation of an assembly-time subroutine that is written in the conditional assembly language of the Assembler. You invoke (or “*call*”) it by writing its name in the operation field of a statement, possibly with a name-field entry, and the arguments to the macro in the operand field. The conditional assembly statements in the macro definition examine the arguments and generate statements in the “ordinary” assembler language that then form part of your program.

There are generally two types of macro argument: *positional*, like the operands of a machine instruction, and *keyword*, of the form name=value. Macro arguments are usually self-defining terms or symbols naming objects or values in your program, or fixed character tokens known to the macro.

The arguments for a macro may also be parenthesized lists, as in Figure 725. For example, you might invoke the OPEN macro this way, using what is called the “Standard” form of the macro:

```
OpenOutS OPEN  (OutFile,OUTPUT)
```

Figure 725. Sample macro invocation, Standard form

In Figure 725, the name field entry is the symbol **OpenOutS**, the macro name in the operation field is **OPEN**, and the (single) argument in the operand field is **(OutFile,OUTPUT)**, where **OUTPUT** is one of several valid fixed tokens.

Because this OPEN macro generates statements that refer to the symbol **OutFile**, we must also define it (this is actually the name of a “Data Control Block”, or DCB, that we'll describe shortly).

In this case, the statements generated by this macro call are:

297 The limit of 256 possibly SVC codes has been increased by the “Extended SVC Routing Facililty”.

```
                                         2 OpenOutS OPEN  (OutFile,OUTPUT)
000000                                   3+         CNOP  0,4                          ALIGN LIST TO FULLWORD
000000 4110 C008           00008         4+OpenOutS LA    1,*+8                        LOAD R1 W/LIST ADR
000004 47F0 C00C           0000C         5+         B     *+8                          BRANCH AROUND LIST
000008 8F                                6+         DC    AL1(143)                     OPTION BYTE
000009 000030                            7+         DC    AL3(OutFile)                 DCB ADDRESS
00000C 0A13                              8+         SVC   19                           ISSUE OPEN SVC
```

Figure 726. Generated statements from an OPEN macro

In Figure 726, statement 2 is the macro invocation. Statements 3 through 8 have a + sign following the statement number; this is the Assembler's indication that the statement is *generated* by conditional assembly, and not part of the original source program. The name field symbol **OpenOutS** was assigned to the first generated statement, and the final generated statement is a "Supervisor Call" instruction with operand "19".

> **Note!**
>
> Most System Interface macros and services save and restore general registers 2-13, but use (and *don't* restore) general registers 0, 1, 14, and 15. Some macros return new values in one or more of those four registers.

Almost all macros that expect the name of a memory location or a length argument will accept an argument with a register number enclosed in parentheses. In this example, the length value (LV=) is currently in GR4, and the address argument (A=) is in GR7.

```
                                        292        FREEMAIN R,LV=(4),A=(7)
000246 1804                             293+       LR    0,4                       LOAD LENGTH
000248 4110 7000           00000        294+       LA    1,0(0,7)                  LOAD AREA ADDRESS
00024C 0A0A                             295+       SVC   10                        ISSUE FREEMAIN SVC
```

The examples in subsequent sections illustrate only a small subset of the options provided by these macros; see the manuals listing in the Bibliography on page 1057 for further information.

Exercises

41.2.1.(1)+ Write short programs that expand into instruction sequences, using the many variations on the SAVE and RETURN macros described in " 37.5. Additional Conventions (*)" on page 777 and study the differences among the generated statements.

41.2.2.(1)+ Identify the types of macro arguments in the FREEMAIN macro above.

41.3. Macro Formats: Standard, List, and Execute

The example in Figure 725 on page 951 uses *Standard* form, in which all generated statements, including parameters to be passed to the Supervisor, are generated directly into the instruction stream, as we saw in Figures 725 and 726. There are two other forms, the *List* and *Execute* forms that don't mix instructions and data like the Standard form. Almost all system service macros support all three forms.

The list form of a macro expansion simply generates the data structures to be passed to the Supervisor, but doesn't generate executable instructions like the LA, branch, and SVC instructions in Figure 726. For example:

```
OpenOutL OPEN  (OutFile,OUTPUT),MF=L
```

Figure 727. Sample macro invocation using List form

The only difference from Figure 725 on page 951 is the additional "keyword" argument, **MF=L** (where **L** means "List form"), that tells the OPEN macro to generate only the parameter list.

```
                                        10 OpenOutL OPEN  (OutFile,OUTPUT),MF=L
000030                                  11+OpenOutL DC    OF'0'                         ALIGN LIST TO FULLWORD
000030 8F                               12+         DC    AL1(143)                      OPTION BYTE
000031 000058                           13+         DC    AL3(OutFile)                  DCB ADDRESS
```

Figure 728. Generated statements from a List form OPEN macro

In Figure 728, only the parameters for the OPEN macro are generated, corresponding to statements 6 and 7 in Figure 726 on page 952. These parameters are then used by an OPEN macro in Execute form.

To write this sample macro in Execute form, we use the added argument **MF=(E,listname)**, where **E** means "Execute form" and the **listname** is the name of the parameter list generated by the List form of the macro.

```
OpenOutE OPEN  MF=(E,OpenList)
```

Figure 729. Sample macro invocation using Execute form

The expansion of this macro completes the set of instructions in statements 4 and 6 generated by the Standard form in Figure 726 on page 952: GR1 points to the parameters, and issues the SVC instruction:

```
                                        15 OpenOutE OPEN  MF=(E,OpenOutL)
000058 4110 C030              00030     17+OpenOutE LA    1,OpenOutL                    LOAD PARAMETER REG 1
00005C 0A13                             18+         SVC   19                            ISSUE OPEN SVC
```

Figure 730. Generated statements from an Execute form OPEN macro

In this case, we omitted the operand describing **OutFile** and that we wanted to open it for **OUTPUT**; the Execute form of the OPEN macro assumes that we have generated the necessary data in the area named **OpenOutL** using the List form as in Figure 728, and simply puts its address in GR1 and issues the SVC.

41.3.1. List form with Empty Argument List

Sometimes we want to use the List form for other OPENs, so we first create an *empty* List form:

```
OpenLstX OPEN  (,),MF=L
```

Figure 731. Sample macro invocation using empty List form

The generated instructions are simple:

```
                                        20 OpenLstX OPEN  (,),MF=L
000080                                  21+OpenLstX DC    OF'0'                         ALIGN LIST TO FULLWORD
000080 80                               22+         DC    AL1(128)                      OPTION BYTE
000081 000000                           23+         DC    AL3(0)                        DCB ADDRESS
```

Figure 732. Generated instructions from empty List form

Then, to use this empty list to insert the name of the file and its input/output status, we write

```
OpenOutE OPEN  (OutFile,OUTPUT),MF=(E,OpenList)
```

Figure 733. Sample macro invocation using Execute form

The macro expansion then inserts the parameter values into the argument list; note that GR14 is used as a work register by the macro in statements 20 and 23.

```
                                        15 OpenOutE OPEN  (OutFile,OUTPUT),MF=(E,OpenLstX)
000058 4110 C030              00030     17+OpenOutE LA    1,OpenLstX                    LOAD PARAMETER REG 1
00005C 94F0 1000        00000           18+         NI    0(1),X'F0'                    CLEAR OPTION 1 BITS
000060 960F 1000        00000           19+         OI    0(1),15                       INSERT OPTION BITS
000064 43E1 0000              00000     20+         IC    14,0(1,0)                     SAVE OPTION BYTE
000068 4100 C098              00098     21+         LA    0,OutFile                     PICK UP DCB ADDRESS
00006C 5001 0000              00000     22+         ST    0,0(1,0)                      STORE INTO LIST
000070 42E1 0000              00000     23+         STC   14,0(1,0)                     RESTORE OPTION BYTE
000074 0A13                             24+         SVC   19                            ISSUE OPEN SVC
```

Figure 734. Generated statements from an Execute form OPEN macro

Because we have specified the original **(OutFile,OUTPUT)** argument, the macro expansion carefully inserts the new information into the empty parameter list at **OpenLstX**, competes the list, and then issues the SVC.

This technique lets you use the same List form to open other data sets, as in

```
OpenOutE OPEN  (InFile,INPUT),MF=(E,OpenList)
```

Figure 735. Another macro invocation using Execute form and same List form

41.3.2. Register Forms and Arguments

The examples of the OPEN macro above require that GR1 contain the address of a parameter list in storage. Some macros, however, pass their arguments entirely in registers, such as some forms of ABEND, GETMAIN, and FREEMAIN. We'll see more examples of these “R-Type” macros in later sections.

For example, we can write an ABEND macro like this:

```
         ABEND 42
```

Figure 736. An R-Type macro invocation generating an argument in a register

Its expansion is:

```
                                  3         ABEND 42
000000                            4+        DS    0H
000000 4110 002A         0002A    6+        LA    1,42                          LOAD PARAMETER REG 1
000004 0A0D                       7+        SVC   13                            LINK TO ABEND ROUTINE
```

Figure 737. Generated statements from R-Type macro

and the only argument is passed directly in GR1, not in memory.

If the address of a macro argument is already in a register, you can provide it directly by parenthesizing the register number, rather than by the name of the argument. For example:

```
         LA    4,INDATA
         LA    9,OUTDATA
         OPEN  ((4),(INPUT),(9),(OUTPUT))   Open two DCBs
```

Figure 738. A macro invocation with arguments in registers

The macro expansion in Figure 739 shows how the addresses in the registers are stored into the generated argument list:

```
000032 4140 F0D0         000D0    35        LA    4,INDATA
000036 4190 F130         00130    36        LA    9,OUTDATA
                                  37        OPEN  ((4),(INPUT),(9),(OUTPUT))   Open two DCBs
00003A 0700                       38+       CNOP  0,4                          ALIGN LIST TO FULLWORD
00003C 4110 F044         00044    39+       LA    1,*+8                        LOAD R1 W/LIST ADR
000040 47F0 F04C         0004C    40+       B     *+12                         BRANCH AROUND LIST
000044 00000000                   41+       DC    A(0)                         OPT BYTE AND DCB ADDR.
000048 00000000                   42+       DC    A(0)                         OPT BYTE AND DCB ADDR.
00004C 5041 0000         00000    43+       ST    4,0(1,0)                     STORE INTO LIST
000050 5091 0004         00004    44+       ST    9,4(1,0)                     STORE INTO LIST
000054 928F 1004      00004       45+       MVI   4(1),143                     MOVE IN OPTION BYTE
000058 0A13                       46+       SVC   19                           ISSUE OPEN SVC
```

Figure 739. Generated statements from a Standard-form macro with arguments in registers

41.3.3. MODE=24, MODE=31

In the expansions of the executable form of the OPEN macro, the parameter list contains a 3-byte address of the Data Control Block (DCB). Sometimes it may be important for some parts of a macro expansion to use 4-byte addresses to refer to items “above the 16MB line”. Using the OPEN macro as an example, we specify an additional argument, **MODE=31**:

```
         OPEN   (OutFile,(OUTPUT)),MODE=31
```

Figure 740. A Standard macro invocation specifying MODE=31

In the macro expansion in Figure 741, you can see in statement 41 that the address of the **OutFile** DCB is 4 bytes long.

```
                                    35         OPEN  (OutFile,(OUTPUT)),MODE=31
000032 0700                         36+        CNOP  0,4                            ALIGN LIST TO HALFWORD
000034 4110 F03C           0003C    37+        LA    1,*+8                          LOAD R1 W/LIST ADR
000038 47F0 F044           00044    38+        B     *+12                           BRANCH AROUND LIST
00003C 8F                           39+        DC    AL1(143)                       OPTION BYTE
00003D 000000                       40+        DC    AL3(0)                         RESERVED
000040 000001A8                     41+        DC    A(OutFile)                     DCB ADDRESS
000044 1801                         42+        LR    0,1                            POINT REGO TO PLIST
000046 1B11                         43+        SR    1,1                            CLEAR REGISTER 1
000048 0A13                         44+        SVC   19                             ISSUE OPEN SVC
```

Figure 741. Generated statements from a Standard-for macro with MODE=31

41.3.4. Mixed Case Macro Arguments

Be careful when using mixed-case characters in macro arguments. Some macros are fussy about the case of their arguments, as the following examples show.

In general, symbols and keyword *parameter* names can be any mixture of upper and lower case; positional and keyword arguments must often be in upper case only.

Compare Figure 742 to Figure 726 on page 952, where the positional argument **OUTPUT** was written in upper case.

```
                                     5 OpenOutS OPEN  (OutFile,Output)
000000                               6+        CNOP  0,4                           ALIGN LIST TO FULLWORD
000000 4110 C008           00008     7+OpenOutS LA    1,+8                          LOAD R1 W/LIST ADR
000004 47F0 C00C           0000C     8+        B     *+8                           BRANCH AROUND LIST
** ASMA254I *** MNOTE ***           10+     12,***  IHB002  INVALID OPTION OPERAND SPECIFIED — Output
```

Figure 742. Example of a mixed-case positional macro argument

In Figure 743, the two keyword arguments LV and A are accepted in lower case; the positional argument R must however be in upper case.

```
                                    13         FREEMAIN  R,lv=(4),a=(7)
000008 1804                         14+        LR    0,4                           LOAD LENGTH
00000A 4110 7000           00000    15+        LA    1,0(0,7)                      LOAD AREA ADDRESS
00000E 0A0A                         16+        SVC   10                            ISSUE FREEMAIN SVC
```

Figure 743. Example of mixed-case keyword macro arguments

In the two macro calls in Figure 744, the two macro-format (MF=) keyword arguments l and e are in lower case, which the macro doesn't recognize:

```
                                    19 OpenOutL OPEN  (OutFile,OUTPUT),mf=l
** ASMA254I *** MNOTE ***           21+     12,***  IHB002  INVALID MF OPERAND SPECIFIED-l
                                    24 OpenOutE OPEN  mf=(e,openoutl)
** ASMA254I *** MNOTE ***           26+     12,***  IHB002  INVALID MF OPERAND SPECIFIED-(e,openoutl)
```

Figure 744. Example of mixed-case keyword macro arguments

41.3.5. The SYSSTATE Macro

On z/OS systems, the SYSSTATE macro can help many system service interface macros to generate correct statement sequences depending on the level of the z/OS operating system. Check the documentation of the SYSSTATE macro to see which arguments and options will generate statements most applicable to your execution environment.

Exercises

41.3.1.(2)+ Assemble the statements in Figure 735 on page 954 (being careful to generate the **OpenList** also), and study the generated statements. Does it matter which of the List and Execute forms of the OPEN macro is generated first?

41.3.2.(1) In Exercise 41.3.1, why is it important to generate **OpenList** also?

41.4. Causing Abnormal Termination

Sometimes a program will find itself in a situation that should never occur, or when continuing could cause more damage than stopping immediately. There are several ways to force termination, including creating program interruptions such as executing an invalid operation code or branching to an invalid address:

```
         - - -                    Discovered an unrecoverable error!
         DC    H'0'               Force program interruption
```

or

```
         DC    X'00nn'            Where 'nn' describes your situation
```

or

```
         LA    1,X'BAD'           Set odd address in GR1
         BR    1                  Force program interruption
```

or

```
         X'C0F4wxyz0002'          Simulate JL *+4; wxyz is your code
```

These will terminate the program unless some error recovery has been set up (see Sections 41.7 and 41.8 starting on page 972) that traps the program interruption. A more common, and more flexible, way to terminate a program is to cause an "abnormal end", or "ABEND"[298] with the ABEND macro.

The ABEND macro supports several useful arguments, including:

- a user code that you provide to indicate the specific termination condition;
- keyword arguments that let you specify whether a memory dump should be provided (you must provide a file where the dump can be written);
- keyword arguments that let you specify whether the entire job step should be ended;
- a reason code that you use to indicate greater detail about the condition causing the termination.

To terminate your program with user code 42, as in Figures 736 and 737 on page 954, you can write

```
         ABEND 42                 End the program immediately
```

As a more complex example, suppose you want to terminate your program with user code 42 and reason code X'BADCODE', and request a memory dump:

```
StopHere ABEND 42,DUMP,REASON=X'BADCODE'    Error in my code!
```

Figure 745. Sample ABEND macro

The macro expansion is shown in Figure 746 on page 957:

298 Uusually pronounced "Ab-End", rather than "A-Bend".

```
                                   3 StopHere ABEND 42,DUMP,REASON=X'BADCODE'    Error in my code!
000000                             4+StopHere DS    0H
000000 4110 004F          0004F    6+         LA    1,42               LOAD PARAMETER REG 1
000004                             7+         CNOP  0,4                ALIGN ON WORD BOUNDARY
000004 47F0 F00C          0000C    8+         B     *+8                BRANCH AROUND CONSTANTS
000008 0BADCODE                    9+         DC    AL4(X'BADCODE')    REASON CODE
00000C 58F0 F008          00008   10+         L     15,*-4             LOAD REG15 WITH REASON CODE
000010 4100 0084          00084   11+         LA    0,132(0,0)         DUMP/STEP/DUMPOPTS/REASON
000014 8900 0018          00018   12+         SLL   0,24(0)            SHIFT TO HIGH ORDER
000018 1610                       13+         OR    1,0                OR IN WITH COMPCODE
00001A 0A0D                       14+         SVC   13                 LINK TO ABEND ROUTINE
```

Figure 746. Generated statements from an ABEND macro

The user code is in the rightmost 12 bits of GR1, the reason code is in GR15, and the high-order byte of GR1 contains bit flags that detail which options are specified and what should be done at termination.

You can also use the DUMPOUT macro for simple memory dumps, without causing program termination. It is described in "Appendix B: Simple I/O Macros" on page 1015.

Exercises

41.4.1.(1)+ What kind of program interruption will be caused by branching to address X'BAD'? Why should you not try to branch to address X'D1E'?

41.4.2(1)+ What kind of program interruption will be caused if you first load GR1 using this instruction?

```
         LGFI  1,C'BAD'
         BR    1                  Force program interruption
```

What if the operand is C'D1E' instead?

41.4.3.(1)+ There's an intentional mistake in Figure 746. What is it?

41.4.4.(2)+ Why does the fourth example at the start of Section 41.4 on page 956 cause a program interruption?

41.5. Storage Management

Many programs need to acquire additional working memory; and the two most commonly used macros are GETMAIN and STORAGE. To release the acquired storage you can use the FREEMAIN and STORAGE macros (with different operands for STORAGE) or simply terminate the program and let the operating system clean up for you.[299]

GETMAIN and FREEMAIN were the original OS/360 storage-management macros, and like STORAGE, are limited to memory below the 2GB "bar". (Storage above the "bar" is managed with the IARV64 macro.)

Take Care!

All three of GETMAIN, FREEMAIN, and STORAGE may use general registers 0, 1, 14, and/or 15 as work registers for various combinations of macro arguments.

[299] But it's much better that your program keep track of the storage it has acquired, and release it when it's no longer needed. There may be a time when your program is part of a larger suite of programs that could possibly exhaust the available storage if all programs don't release what they acquire.

41.5.1. The GETMAIN Macro

GETMAIN provides several ways to request storage:

Conditional (C)
If the requested storage is available, GETMAIN will put zero in GR15; otherwise a nonzero value is returned in GR15.

Unconditional (U)
If the requested storage is not available, GETMAIN will terminate the program with an ABEND.

Register (R)
A request for a single block of storage. Parameters are passed only in registers; some of them may have been constructed in storage and then loaded into registers. (This form is often used during program initialization to allocate working storage.)

Element (E)
A request for a single block of storage.

Variable (V)
A request for a single block of storage between a minimum and a maximum size.

List (L)
A request for several blocks of storage.

Some of these can be combined; for example, **VRC** means that you want to acquire storage between two limits (**V**), the arguments will be passed in registers (**R**), and the system puts a return code in GR15 to indicate the success or failure of the request (**C**).

The supported combinations that can be requested by the first (positional) operand of GETMAIN are summarized in Table 434:

	Single		**Variable**		
Type	**Conditional**	**Unconditional**	**Conditional**	**Unconditional**	**Where**
Register, Single Element		R			<16MB
Register, Single Element	RC	RU	VRC	VRU	<2GB
Single Element	EC	EU	VC	VU	<16MB
List of Elements	LC	LU			<16MB

Table 434. GETMAIN request options

As shown in the last column of Table 434, storage acquired using any of the Register forms *except* **R** can be requested either below or above the 16MB "line", and all addresses and lengths have 31-bit lengths. For the R form, storage must be below the 16MB "line", and addresses and lengths are at most 24 bits long. For the other Register forms, the storage is allocated in the same area of memory as the calling program (that is, below or above the 16MB "line"). For all other forms, the storage is allocated below 16MB.

The address of the acquired storage is returned in GG1, with bits 0-32 set to zero.

Acquired storage is always aligned on a doubleword boundary, with length that is a multiple of 8 bytes. Sometimes, you can request page alignment on a 4K boundary.

Several forms require that you provide one or more words in memory into which lengths and/or addresses will be stored; be careful not to modify them during program execution, because they can be used to release the acquired storage using the FREEMAIN macro.

Figure 747 shows an example of a typical GETMAIN request:

```
         GETMAIN R,LV=72          Get storage for local save area
```

Figure 747. Sample R=type GETMAIN request

The expansion of this macro call is shown in Figure 748 on page 959; note that the only value passed to SVC 10 is in GR0. (The BAL instruction sets the high-order bit of GR1 to 1; we'll see why this is interesting when we discuss the FREEMAIN macro.)

```
                                 5         GETMAIN R,LV=72          Get storage for local save area
000000 4100 07D0        007D0    6+        LA    0,72(0,0)          LOAD LENGTH
000004 4510 C008        00008    7+        BAL   1,*+4              INDICATE GETMAIN
000008 0A0A                      8+        SVC   10                 ISSUE GETMAIN SVC
```

Figure 748. Expansion of a sample R-type GETMAIN request

For the R-type Register form, storage will be allocated below the 16MB "line". For the other forms, storage will be allocated by default in the area where the requesting program is executing, either below or above the 16MB "line".

Figure 749 shows an example of a VRU request:

```
                                 34        GETMAIN VRU,LV=(144,72)
00003C                           35+       CNOP  0,4                ALIGN DATA ON FULLWORD BDY
00003C 47F0 C04C        0004C    36+       B     *+16-4*0-4*0-2*0   BRANCH PAST DATA
000040 00000090                  37+       DC    A(144)             MAXIMUM LENGTH
000044 00000048                  38+       DC    A(72)              MINIMUM LENGTH
000048 00                        39+IHB0004F DC  AL1(0)             RESERVED
000049 00                        40+       DC    AL1(0)             RESERVED
00004A 00                        41+       DC    AL1(0)             SUBPOOL
00004B 0A                        42+       DC    BL1'00001010'      MODE BYTE
00004C 9801 C040        00040    43+       LM    0,1,*-12+2*0       LOAD MAX AND MIN LENGTHS
000050 58F0 C048        00048    44+       L     15,IHB0004F        LOAD GETMAIN PARMS
000054 0A78                      45+       SVC   120                ISSUE GETMAIN SVC
```

Figure 749. Expansion of a sample VRU-type GETMAIN request

In this case, GR0, GR1, and GR15 all contain values used by SVC 120.

For all the Register forms, the address of the allocated storage is returned in GG1, and the high-order bits 0-31 are zero.

If you GETMAIN 8192 or more bytes, or a multiple of 4096 bytes on a page boundary, the system will automatically clear the area to zeros.[300]

41.5.2. The FREEMAIN Macro

The FREEMAIN macro is used to release storage acquired by GETMAIN. For single elements, the values passed to the SVC routine are the length and address of the area to be freed. FREEMAIN supports forms like those of GETMAIN:

Conditional (`C`)
: If the requested storage can be freed, FREEMAIN will put zero in GR15; otherwise a nonzero value is in GR15.

Unconditional (`U`)
: If the requested storage can't be freed, FREEMAIN will terminate the program with an ABEND.

Register (`R`)
: A request to return a single block of storage. Parameters are passed in registers; some of them may be constructed in storage and then loaded into registers.

Element (`E`) and Variable (`V`)
: A request to free a single block of storage. Because both `E` and `V` GETMAIN requests allocate a single area of storage, the equivalent FREEMAIN forms release a single fixed-length area. (The GETMAIN and FREEMAIN argument lists are different.)

List (`L`)
: A request to free one or more blocks of storage.

[300] With some rather specialized restrictions; see the reference manuals in the Bibliography.

The supported combinations that can be requested by the first (positional) operand of FREEMAIN are summarized in Table 435 on page 960:

Type	Conditional	Unconditional	Where
Register, Single Element		R	<16MB
Register, Single Element	RC	RU	<2GB
Single Element	EC, VC	E, EU; V, VU	<16MB
List of Elements	LC	L, LU	<16MB

Table 435. FREEMAIN request options

As the last column of Table 435 indicates, the R form can free storage only below the 16MB "line". The **RC** and **RU** Register forms can free storage allocated either above or below the 16MB "line". All the other forms can free storage only below 16MB.

Figure 750 shows a simple example of a typical FREEMAIN request:

```
                                   60         FREEMAIN R,LV=72,A=ZAddr
000072 4100 0048          00048    61+        LA    0,72(0,0)                      LOAD LENGTH
000076 5810 C084          00084    62+        L     1,ZAddr                        LOAD AREA ADDRESS
00007A 0A0A                        63+        SVC   10                             ISSUE FREEMAIN SVC
```

Figure 750. Example of an R-type FREEMAIN macro

In this case, the length is loaded into GR0 and the address is put in GR1. Note that the *same* SVC 10 is used as for GETMAIN; the key difference here is that the high-order bit of GR1 is zero (because the address of the acquired storage is below 16MB), while it was 1 for GETMAIN.[301]

41.5.3. The STORAGE Macro

The STORAGE macro combines the functions of acquiring and releasing storage in one macro, using the positional arguments OBTAIN and RELEASE to select the operation. To specify conditional and unconditional operations, you must specify COND=YES or COND=NO, respectively; NO is the default.

Figure 751 shows a simple request for 72 bytes using the STORAGE macro.

```
         STORAGE OBTAIN,LENGTH=72      Request 72 bytes
```

Figure 751. Sample STORAGE OBTAIN request

The instructions generated from this request are shown in Figure 752 on page 961. (Note that registers 0, 14, and 15 are used; compare this expansion to the equivalent request using GETMAIN in Figure 748 on page 959.)

[301] The BAL instruction in Figure 748 on page 959 *is* doing something important!

```
                                    4          STORAGE OBTAIN,LENGTH=72      Request 72 bytes
000000                              5+         CNOP   0,4
000000 47F0 F00C          0000C     6+         B      IHB0001B               .BRANCH AROUND DATA
000004 00000048                     7+IHB0001L DC     A(72)                  .STORAGE LENGTH
000008 00                           8+IHB0001F DC     BL1'00000000'
000009 00                           9+         DC     AL1(0*16)              .KEY
00000A 00                          10+         DC     AL1(0)                 .SUBPOOL
00000B 02                          11+         DC     BL1'00000010'          .FLAGS
00000C                             12+IHB0001B DS     0F
00000C 5800 F004          00004    13+         L      0,IHB0001L             .STORAGE LENGTH
000010 58F0 F008          00008    14+         L      15,IHB0001F            .CONTROL INFORMATION
000014 58E0 0010          00010    15+         L      14,16(0,0)             .CVT ADDRESS
000018 58EE 0304          00304    16+         L      14,772(14,0)           .ADDR SYST LINKAGE TABLE
00001C 58EE 00A0          000A0    17+         L      14,160(14,0)           .OBTAIN LX/EX FOR OBTAIN
000020 B218 E000    00000          18+         PC     0(14)                  .PC TO STORAGE RTN
```

Figure 752. Example of a STORAGE OBTAIN macro expansion

If the request was successful, the address of the obtained storage is returned in GG1.

To return the 72 bytes requested previously, we assume that their address is now in GR1, as indicated by the ADDR=(1) operand in Figure 753.

```
         STORAGE RELEASE,LENGTH=72,ADDR=(1) Return 72 bytes
```

Figure 753. Sample STORAGE RELEASE request

The instructions generated by this macro are shown in Figure 754 (again, compare this to the equivalent FREEMAIN in Figure 750 on page 960).

```
                                   20          STORAGE RELEASE,LENGTH=72,ADDR=(1) Return 72 bytes
000024                             21+         CNOP   0,4
000024 47F0 F030          00030    22+         B      IHB0003B               .BRANCH AROUND DATA
000028 00000048                    23+IHB0003L DC     A(72)                  .STORAGE LENGTH
00002C 00                          24+IHB0003F DC     BL1'00000000'
00002D 00                          25+         DC     AL1(0*16)              .KEY
00002E 00                          26+         DC     AL1(0)                 .SUBPOOL
00002F 03                          27+         DC     BL1'00000011'          .FLAGS
000030                             28+IHB0003B DS     0F
000030 5800 F028          00028    29+         L      0,IHB0003L             .STORAGE LENGTH
000034 58F0 F02C          0002C    30+         L      15,IHB0003F            .CONTROL INFORMATION
000038 58E0 0010          00010    31+         L      14,16(0,0)             .CVT ADDRESS
00003C 58EE 0304          00304    32+         L      14,772(14,0)           .ADDR SYST LINKAGE TABLE
000040 58EE 00CC          000CC    33+         L      14,204(14,0)           .OBTAIN LX/EX FOR RELEASE
000044 B218 E000    00000          34+         PC     0(14)                  .PC TO STORAGE RTN
```

Figure 754. Example of a STORAGE RELEASE macro expansion

The STORAGE macro supports a greater variety of options compared to GETMAIN and FREEMAIN. Note also that it generates a PC instruction rather than the SVCs used by GETMAIN and FREEMAIN.

41.5.4. Subpools (*)

The three macros described above support *subpools*. A subpool is a way to group related requests. For example, if your program is creating variable-sized binary trees, linked lists, and tables, you may need to allocate additional storage for each as the need arises. By allocating each type in a separate subpool, you can then release all the storage allocated for one of those functions at once, without having to keep track of each separately allocated segment.

Subpools (and their many types) are described in the references listed in the Bibliography on page 1057.

41.5.5. Optional Operands (*)

The GETMAIN and STORAGE macros have many additional optional operands such as LOC= (which lets you specify where relative to the 16MB "line" the allocated storage should be). See the relevant manuals in the Bibliography.

Exercises

41.5.1.(1)+ Why does the BAL instruction in Figure 748 on page 959 always generate a high-order 1 bit in GR1, whether executed in 24-bit or 31-bit addressing mode?

41.5.2.(3) It was stated (following Figure 748 on page 959) that for the conditional and unconditional forms of GETMAIN, the storage is allocated by default by the GETMAIN macro in the area where the requesting program is executing, whether above or below the 16MB "line". By reading the documentation for GETMAIN and STORAGE, determine how a program executing below the 16MB "line" can request storage above the 16MB "line".

41.6. Basic Input and Output

Input and Output ("I/O" for short) is a vast and complex subject, many aspects of which are well beyond the scope of this text. We will introduce key features used by many application programs.

41.6.1. A Simple Scenario

We'll start with a simple scenario in six basic steps, showing the key actions involved in your program's reading records from a Data Set. The figures are expanded in detail at each step.

Simple Scenario, Step 1: The Data Set

The Data Set is on some external storage medium such as tape or disk. Almost all Data Sets have some type of *label* describing the properties of the records in the Data Set. A part of the label is the *Data Set Name* (*DSName*) that you can use to refer to the Data Set.

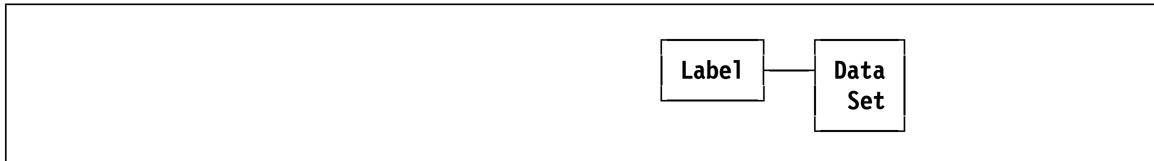


Figure 755. A Data Set with records you want to read

You have written a program to read the records, and submit a job to the operating system to load and execute the program.

Simple Scenario, Step 2: Submitting a job to execute the program

When you submit the job to read from the Data Set, you specify in the *Job Control Language* (JCL) a *Data Definition Name* (DDName) that your program will need to access the data. The operating system Supervisor creates a *Job File Control Block* (JFCB) from information in the JCL statements you provided.

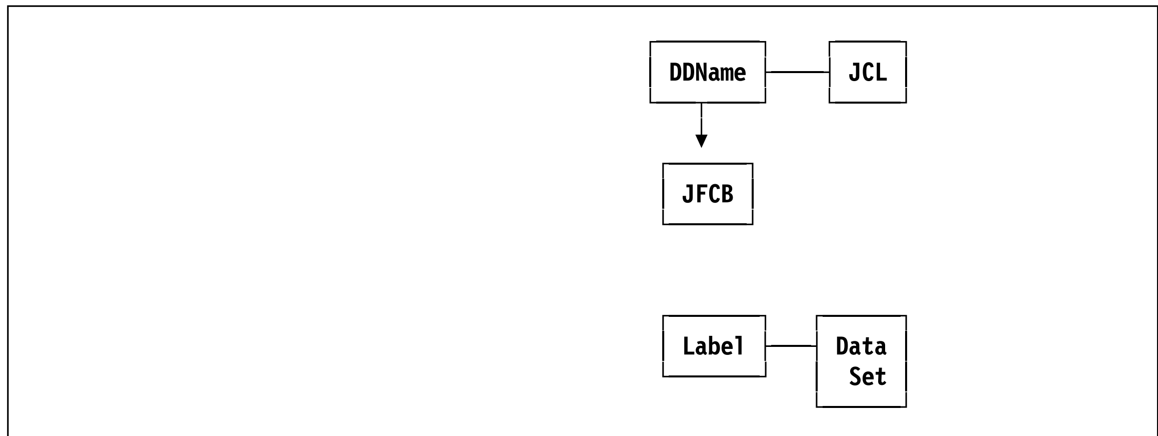


Figure 756. You submitted a job with a program to read the records

After the Supervisor processes the JCL statements, your program is ready to be loaded.

Simple Scenario, Step 3: Your program after loading and before execution

Your program has been loaded into memory. It contains the *Data Control Block* (DCB) you wrote; it has information about how your program will access the records, as well as your OPEN, GET, and CLOSE macros, and a Record Buffer area where the records will be placed as they're read.

- The OPEN macro, when executed, gives you access to your data.
- The GET macro requests a data record be read.
- The CLOSE macro terminates access to the Data Set.

The DCB contains the DDName that provides access to the Data Set whose DDName (now saved in the JFCB) was specified in JCL statements.

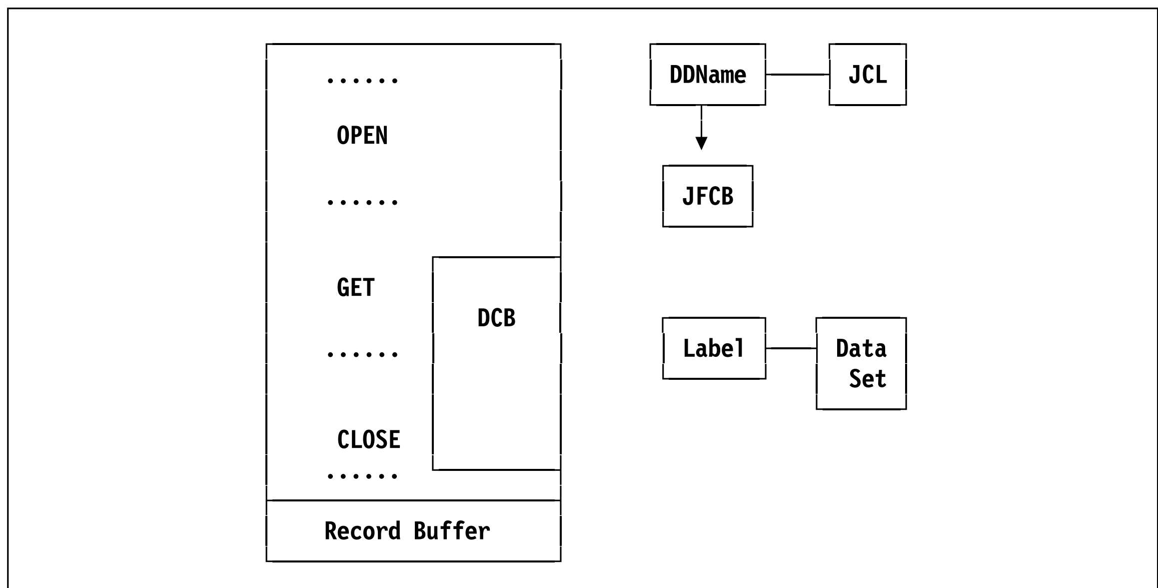


Figure 757. Your program, loaded into memory before execution

Your program is ready to start execution.

Simple Scenario, Step 4: OPENing the data set

After establishing addressability and initializing, your program executes the OPEN macro; many things happen.

1. Information from the JFCB, the Data Set Label, and from the DCB is merged into the DCB, as described in Section 41.6.5 on page 969.
2. *Access Method* routines appropriate to the type of device holding the Data Set, the type of records, the type of buffering, etc., are loaded automatically. These routines provide *device independence* to your program.

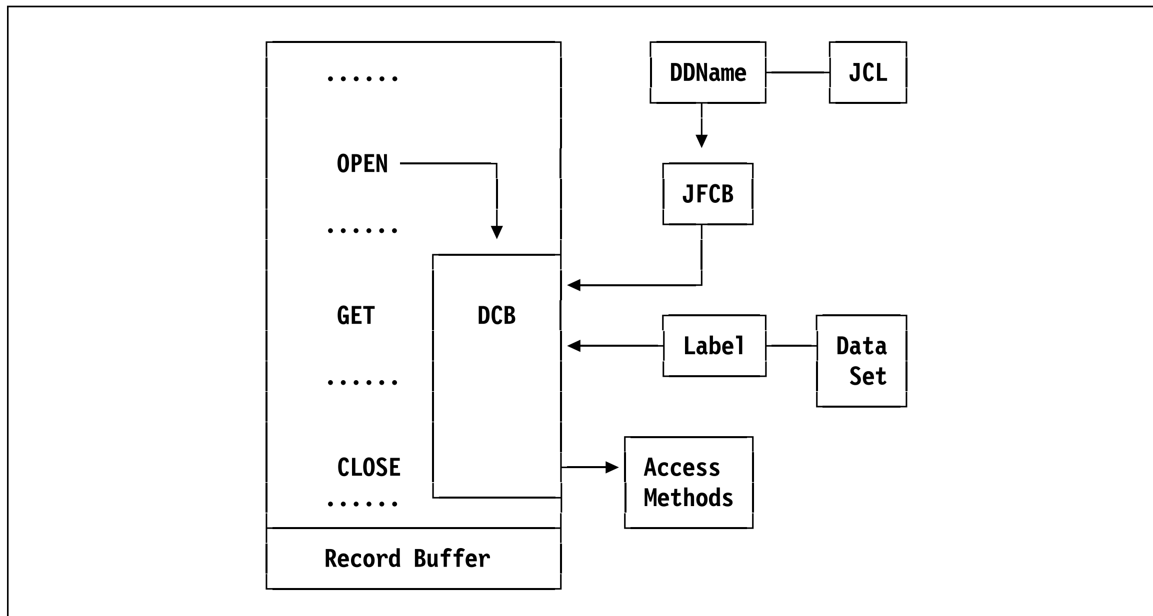


Figure 758. Your program after executing the OPEN macro

Many things can go wrong at this stage, typically causing an ABEND with system completion code x13, where "x" is a hexadecimal digit, and "13" is the hexadecimal SVC number (noted as **I** in Table 432 on page 950). Most system ABEND codes use this convention.

Simple Scenario, Step 5: GETting a record

The GET macro points to the DCB and to the Record Buffer, and calls the Access Method routines to read a record from the Data Set and place it in the Record Buffer. For example, you might write

```
GET   MyDCB,RecBuf       Read a record from MyDCB into RecBuf
```

If you specified an end-of-data address (EODAD, described below) in the DCB, control goes there when there are no more records in the Data Set.

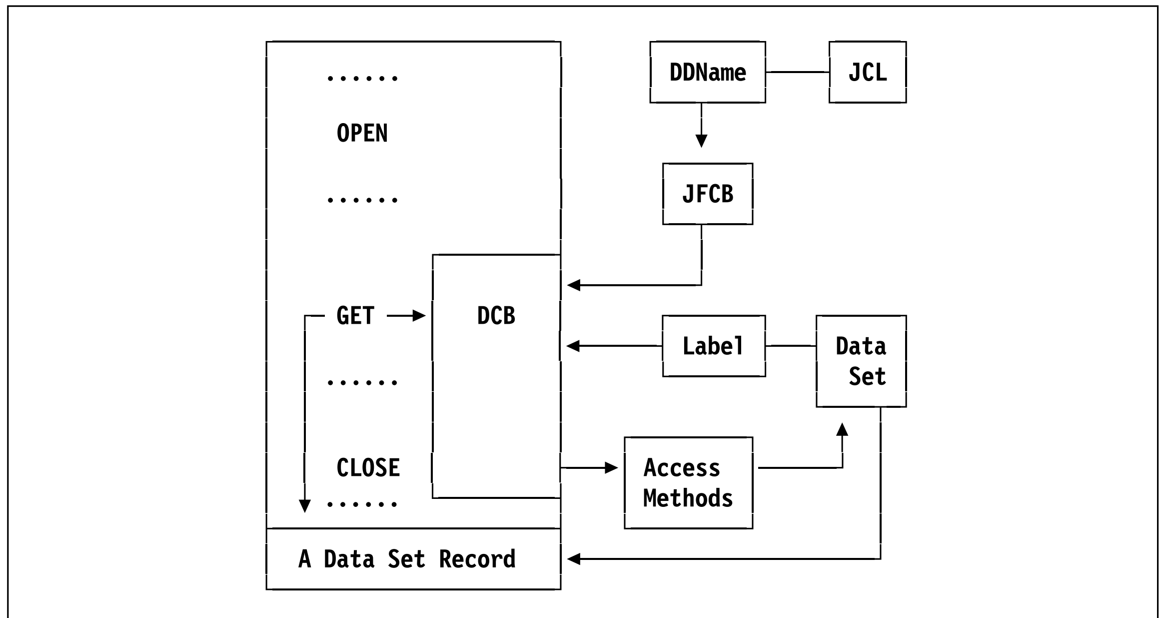


Figure 759. Your program after executing the GET macro

Simple Scenario, Step 6: CLOSEing the DCB

When you're finished, you execute a CLOSE macro: it specifies the name of the DCB you want to close. Your DCB is restored to its initial status; the Data Set Label is updated if necessary (if you wrote to the Data Set), and the updated JFCB is retained in case you want to re-OPEN the Data Set.

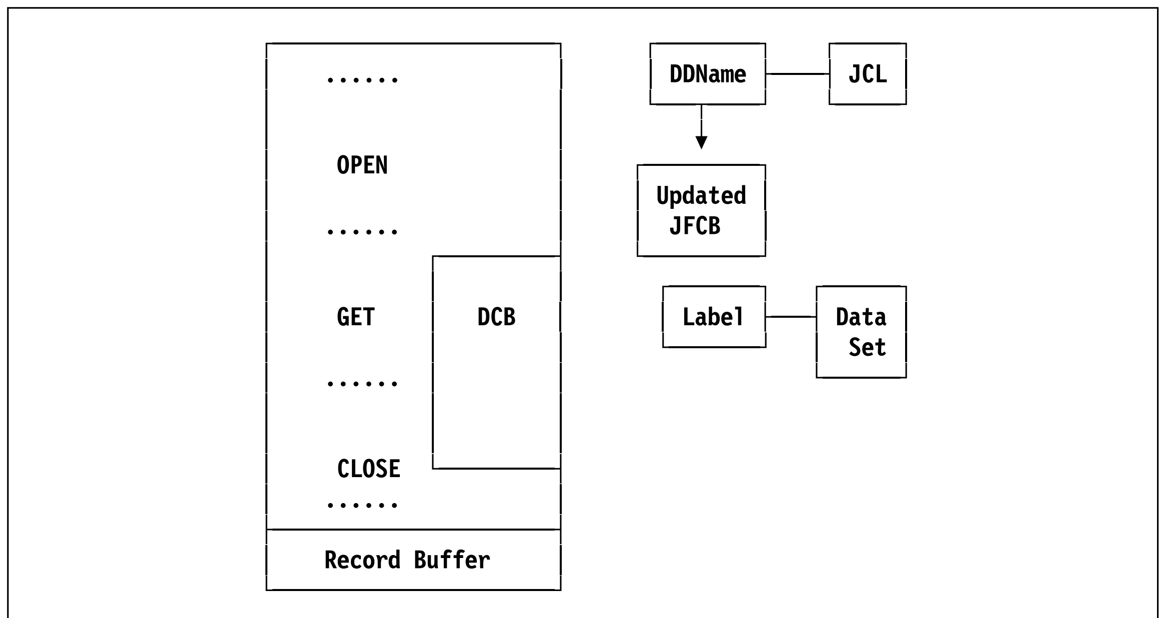


Figure 760. Your program after executing the CLOSE macro

Your program has been restored (almost) to its initial state, except for changes you may have made to data fields and work areas. The program then tidies up, and returns to the system.

This brief scenario should help you follow the details that follow.

41.6.2. Access Techniques and Access Methods

There are many access techniques available for data. Data items can be written or read sequentially (one after the other); directly (put the item in a specific position in a data set, or read it from that position, ignoring other items); by index (a field in the data is used to identify it), and so on. Each of these can be very complex; we will examine only sequential access, and only in its simple forms.

There are two main ways to read or write data sequentially:

Basic You manage *blocks* of data, independent of any internal structure. Each block may have multiple records; you must organize the data items in each block. Each I/O operation is initiated by a READ or WRITE macro, and you must synchronize the I/O operations with read or write completion using the CHECK and WAIT macros.

Queued You manage individual data *records*. The Supervisor does almost everything for you: it handles I/O operations, synchronization of I/O activity and data availability, blocking and deblocking records, etc. You read and write records with macros like GET and PUT.

We will describe only the simplest form, queued access.

There are several ways to manage records using queued access; the simplest forms are called "BSAM" (Basic Sequential Access Method) and "QSAM" (Queued Sequential Access Method).[302] Depending on values in your DCB, the system will load appropriate "access method routines" to support the various types of data movement.

With QSAM, you can choose one of the following modes (among others):

Move mode
Your GET macro points to your buffer where the system should place your input record; or, your PUT macro points to your buffer from which the output record should be written by the system.

Locate mode
Your GET macro returns a pointer to where your input record can be found; your PUT macro returns a pointer to where your output record should be placed.

In practice, QSAM is far easier to use than BSAM; Table 436 illustrates some differences.

QSAM	BSAM
• Supports all record formats • Your interface is "logical records" • Automatic blocking and deblocking • Automatic buffer management • Automatic I/O operation synchronization	• Supports all record formats • Your interface is "physical blocks" • Blocking and deblocking is your problem • Buffer management is your problem • Synchronizing I/O operations with use of the data is your problem

Table 436. Comparing QSAM and BSAM

41.6.3. The Data Control Block (DCB)

The DCB macro is a complex structure, with nearly 100 parameters (almost all of which you won't care about), some of which can have as many as 30 valid values (almost all of which you won't care about). It is a part of your program. Some values in the DCB can come from other sources, such as JCL DD statements, a data set label, and from your program during execution. All fields must be completed by the end of the OPEN process, before I/O operations can begin.

The DCB contains 3-byte address fields such as the address of the access method routines and the EODAD and EXLST addresses described below, so it must reside below the 16MB "line".

302 Pronounced "Bee-Sam" and "Cue-Sam".

You *must* specify the DSORG (Data Set Organization) and MACRF (Macro Format) arguments at assembly time. If you intend to provide a DCB OPEN exit to examine and/or modify the DCB during the OPEN process, you must specify the EXLST argument. Values for other arguments can be supplied during OPEN by the JCL DD statement, the data set label, and the OPEN exit.

DSORG Must be coded in the DCB macro; the most usual values are:

- **PS** Physical Sequential: strictly sequential access. Both of the following are defined when you specify PS:
  - **BS** Basic Sequential: strictly sequential access.
  - **QS** Queued Sequential: strictly sequential access.
- **PO** Partitioned Organization, for libraries, which are collections of sequential data sets identified by a *member name* that's contained in a directory. All members have the same characteristics, and are accessed sequentially. The access technique for members of a library is called "BPAM"[303] (Basic Partitioned Access Method); members are located with the FIND macro, and read and written with READ and WRITE macros.

If you code more than one DSORG value, you can re-OPEN the DCB for a different type of I/O after closing for the previous use.

MACRF Must be coded in the DCB macro; don't omit it.[304] These QSAM forms are the simplest to use:

- **GM** Get Move: Logical record processing, move mode input. The access method routines read blocks of records into internal buffers, unblocks your record for you, and moves it to the work area you defined as the second operand of the GET macro. Your work area must be at least as long as the record!
- **PM** Put Move: Logical record processing, move mode output. You tell the PUT macro where your record is; the system moves it to its internal buffers; a buffer is output when it is full.
- **GL** Get Locate: Logical record processing, locate mode input. The access method tells you where in its internal buffers the record is found.
- **PL** Put Locate: Logical record processing, locate mode output. The access method tells you where to put your record in its internal buffers.

These are some other arguments that can be supplied during initialization or execution.

DDNAME Can be coded in the DCB macro, or supplied by the program before OPEN

RECFM Figures 762-764 starting on page 968 illustrate the following record formats.

- **F** Fixed length unblocked records (BLKSIZE=LRECL)
- **FB** Fixed length blocked records (BLKSIZE=n×LRECL)
- **V** Variable length unblocked records
- **VB** Variable length blocked records
- **U** Undefined ("None of the above")
- **xxA** Records have ANSI carriage-control characters in the first byte

LRECL The length of each F record, or the maximum length for V records. Often omitted for input data sets; it can be supplied from the data set label at OPEN time. The LRECL field is updated after each read of a record if it changes value.

BLKSIZE The maximum length of a block; for FB, it must be a multiple of the LRECL value; for VB, the maximum block length.

---

303 Pronounced "Bee-Pam".

304 If you omit a MACRF value, the assembly will assign the default E (with severity 8), which means (if you proceed) that you must write your own device-dependent channel programs and do everything yourself. It's quite difficult.

EODAD For reads, the address to receive control on End of Data (EOD). The EOD routine can either return to the address in GR14 on entry, or continue normal processing (this is the most common way). If no EODAD address is provided at end of data, or if you try to read after the end of data, the system will terminate your program with a System 337 ABEND.

The 3-byte address of the EODAD routine means that it will be entered in 24-bit addressing mode, so it must reside below the 16MB "line". It can of course change addressing modes so long as it changes back to 24-bit mode before returning to the system.

EXLST The exit list specifies special exit routines; the DCB OPEN exit is the most most usual, for which you specify the name of a word-aligned list of the form `X'85',AL3(OPEN-exit_routine)`. The DCB OPEN exit routine can modify or complete a DCB before the data set is completely open.

When it receives control, the low-order 3 bytes of GR1 point to the DCB (remember that the DCB and the OPEN exit must be in 24-bit storage). GR14 must be preserved and used to return to finish OPEN processing. You must not use the address in GR13 for a save area, and you need not preserve the contents of GRs 2-13 (the system does that for you).

A typical DCB statement might look like this:

```
dcbname   DCB  DDNAME=xxxxxxxx,        To match up with the JCL DDName      X
               MACRF=xx,               Macro Format: how to process records X
               DSORG=xx,               Data Set Organization                X
               LRECL=nnn,              Logical Record Length                X
               BLKSIZE=nnnnn,          For blocked records                  X
               RECFM=xx,               Record Format                        X
               EODAD=xxxxxxx           End-of-data address for input
*                                                                column 72 ↑
```

Figure 761. Example of typical DCB parameters

41.6.4. Important Record Formats

- Fixed: all records have the same length, and may be grouped into *blocks* containing (usually) a fixed number of records. It's best to have no truncated (partially filled) blocks except for the last block.

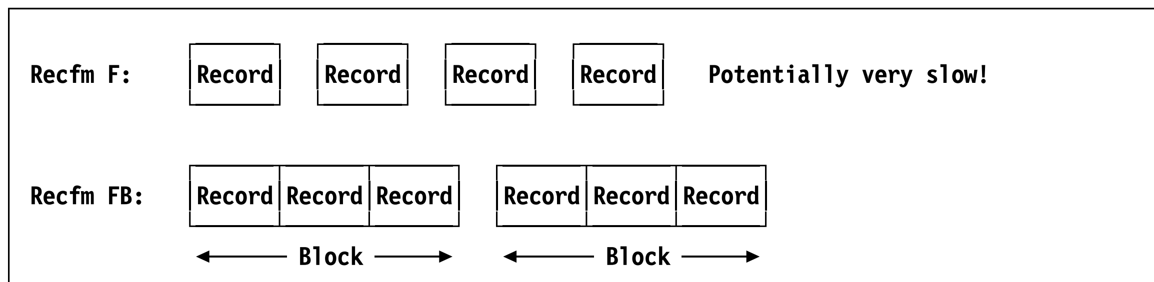


Figure 762. Unblocked and blocked F-type record and block formats

- Variable: each record is preceded by a 4-byte Record Descriptor Word (RDW) giving the record length in the first two bytes; that length *includes* the 4-byte length of the RDW. V-format records may be blocked; a block of records is preceded by a 4-byte Block Descriptor Word (BDW) giving the length of the block (including its own length) in the first two bytes. For both the BDW and RDW, the remaining two bytes must be zero.

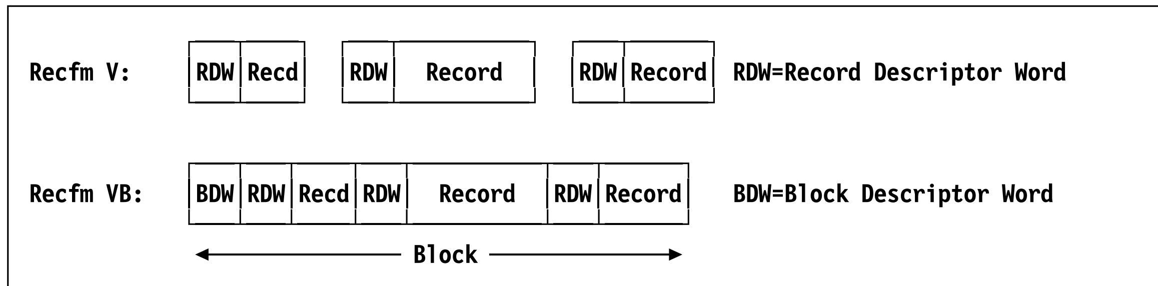


Figure 763. Unblocked and blocked V-type record and block formats

- Undefined: A single physical record; do any blocking/unblocking (if needed) yourself. You must understand the internal structure of U-format blocks that you read or write.

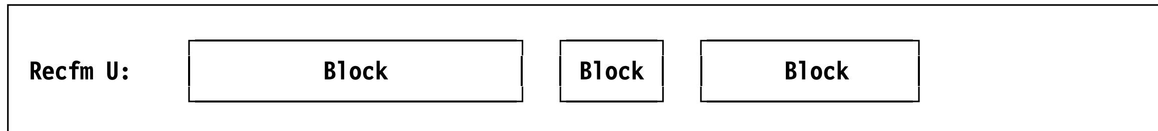


Figure 764. U-type block formats

41.6.5. Opening the DCB

Figure 765 and the following explanations outline the steps taken during initialization of a DCB by OPEN:

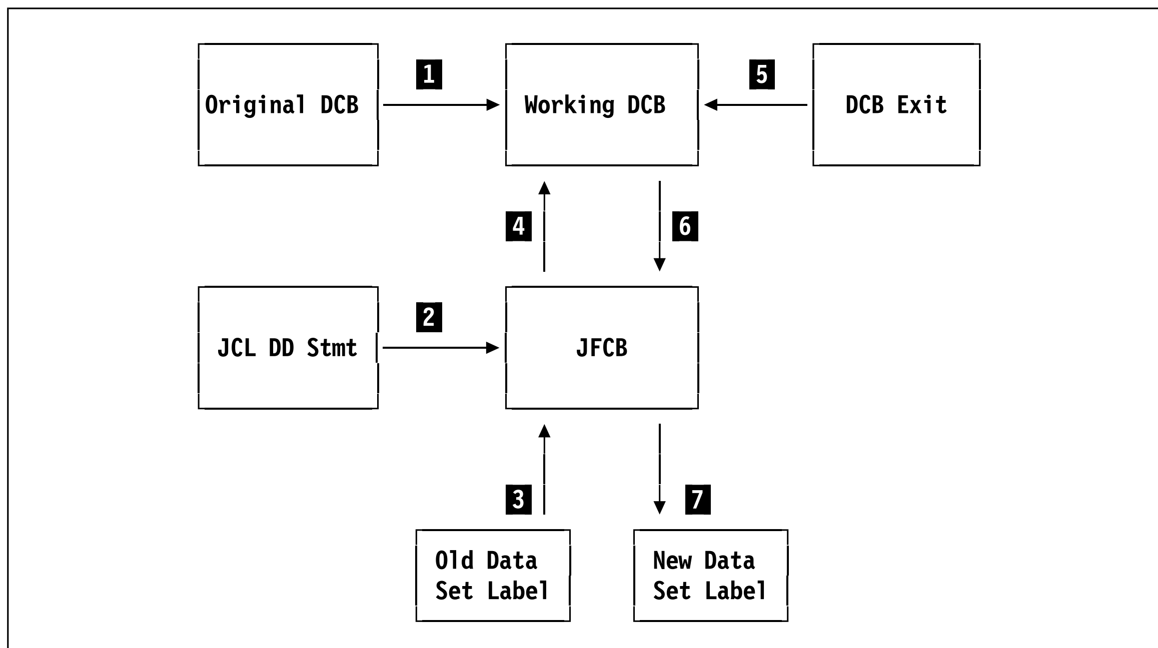


Figure 765. Completion of a DCB during OPEN processing

1 The settings of the original DCB are saved so they can be restored when the DCB is closed.

2 Data from the JCL DD statement was used to create the Job File Control Block (JFCB).

3 Fields in the JFCB not already completed are filled from the data set label.

4 Fields in the DCB not already completed are filled from the JFCB.

5 If a DCB OPEN exit exists, it is given control to make any desired modifications to the DCB.

6 The merged and updated fields from the DCB are copied back into the JFCB.

7 If the data set has been opened for output, fields from the JFCB are used to create the new data set label.

OPEN then chooses and loads access method routines according to your choices of DSORG, buffering technique, access technique, and device type. If OPEN fails, you normally get a system 013 ABEND.

41.6.6. Closing the DCB

To close a DCB and terminate I/O to that DDName, simply write

```
         CLOSE dcbname
```

For the QSAM macros we've seen, you should follow that with

```
         FREEPOOL dcbname
```

so the access method routines will free the storage they acquired for internal buffers.

41.6.7. The DCBD Macro and the IHADCB Dummy Section

The DCBD macro generates symbolic names for all DCB fields, in the IHADCB Dummy Control Section. You specify at most two arguments:

```
         DCBD  DSORG=xx,              Data Set Organization(s)            X
               DEVD=xx                Device type(s)
```

Figure 766. DCBD operands

DSORG Types of access for which your DCBs were written.

DEVD Specifies the types of devices to be included.

However, if you write

```
         DCBD  DSORG=(PS,PO)          Data Set Organization(s)
```

Figure 767. DCBD operands

then the presence of the PS argument means you can omit the DEVD argument entirely.

If you code multiple DSORG values, you can use IHADCB[305] to refer to multiple DCBs; see Figure 768.

Important fields defined in the IHADCB DSect are the DCBOFLGS byte, which contains the DCBOFOPN bit you should test to verify that the DCB OPENed correctly, and the DCBLRECL field that you must use when reading and writing V- and U-format records.

For example, you can use the IHADCB dummy section to map each DCB in your program, with the help of Labeled Dependent USING statements:

```
         USING Program,12         Using statement for entire program
         - - -
         OPEN  (SYSINDCB,INPUT,SYSOUDCB,OUTPUT)
Sysin    USING IHADCB,SYSINDCB    Map SYSIN DCB with IHADCB
         TM    Sysin.DCBOFLGS,DCBOFOPN  Test if OPEN was successful
         JZ    BadSysin           Go take recovery action
Sysou    USING IHADCB,SYSOUDCB    Map SYSOUT DCB with IHADCB
         TM    Sysou.DCBOFLGS,DCBOFOPN  Test if OPEN was successful
         JZ    BadSysou           Go take recovery action
```

Figure 768. Using IHADCB to map two different DCBs simultaneously

[305] Sometimes known as the "I Had A Control Block" macro.

The two DCBs are within the range of the USING statement; the two Labeled Dependent USING statements let you map each DCB with the same IHADCB DSECT.

41.6.8. The DCBE Macro and 31-bit Address Mode

All the I/O examples we've seen were required to execute in 24-bit addressing mode, and therefore reside below the 16MB “line”. There are other macros (like DCBE, the *DCB Extension*) that let you put things like buffers and exit routines above the line. This can be important if available storage below the line is limited. To learn more, see the “Data Sets” references in the Bibliography on page 1057.

41.6.9. I/O Summary

This has been a *very* simple overview of a vast and complex subject. z/OS supports the richest variety of facilities among modern operating systems, while some other operating systems have extremely simple I/O models. Most programs need only a small and useful subset of all the capabilities z/OS offers.

Why use BSAM? If you have large volumes of data in which records can be processed independently, you can assign parallel tasks to process the records in each block as it is read. This can give higher rates of “throughput” than QSAM could support, because QSAM handles one record at a time.

Another popular access method is the “Virtual Storage Access Method” (VSAM), which supports sequential, direct, and indexed access. It requires different macros and control, and its flexibility and complexity are beyond the scope of this text.

Some general guidelines:

- Specify INPUT or OUTPUT (etc.) on the OPEN macro.
- Omit device-dependent parameters and macros when possible, to allow for varied devices, record formats, etc. at execution time. Put device-dependent and data-set-specific information on the DD statement.

  Don't code more in the DCB than is required to ensure correct processing. Other parameters can be completed during initialization or execution.[306]
- Avoid unblocked records: there is a potential performance penalty in time and CPU costs.[307]
- A DCB can be used for multiple data sets so long as it is closed before opening for a different data set. You may need to modify some DCB fields if any there are differences in data set characteristics. You *will* need to modify the DDNAME field!

41.6.10. A Sample Program

This sample program uses the macros described above to write a printable record.

[306] Some systems will assign an “optimum” BLKSIZE for output data sets depending on its and the receiving device's physical properties, and for input data sets based on its existing BLKSIZE.

[307] I once saw a case where a program reading unblocked 80-byte records took nearly 10 minutes of elapsed clock time, but completed in about 10 seconds when the input data set was adequately blocked. (It was an old, slow machine.)

```
Sample   CSect ,
Sample   AMode 24
Sample   RMode 24
         STM   14,12,12(13)        Save caller's registers
         LR    12,15               Copy base register
         Using Sample,12           Establish addressability
         LR    2,13                Copy caller's save area address
         LA    13,Save             Point to local save area
         ST    2,Save+4            Store back chain
         ST    13,8(,2)            Store forward chain
         OPEN  (PrintDCB,(OUTPUT))  Open the print DCB
         PUT   PrintDCB,Line       Output the message
         CLOSE PrintDCB            Close the print DCB
         FREEPOOL PrintDCB         Release buffers
         L     13,Save+4           Restore caller's save area address
         RETURN (14,12)            Restore registers
         BR    14                  Return
Line     DC    CL121'1Greetings from a sample program'
         Print NoGen               Suppress DCB details
PrintDCB DCB   DSORG=PS,MACRF=PM,LRECL=121,BLKSIZE=121,RECFM=FA,       X
               DDNAME=PRINT
Save     DC    9D'0'               Local save area
         End
```

Figure 769. A complete sample program

The output of this little program is:

```
1Greetings from a sample program
```

41.7. Handling Program Interruptions

Programs can handle their own exception conditions on two levels:

1. Program interruptions can be investigated using a “Program Interruption Exit”; in the absence of an exit, the system will generate an ABEND using SVC 13, as illustrated in Figure 746 on page 957.
2. Abnormal terminations (ABENDs) can be investigated using a “Task Abnormal Termination Exit” that receives control on an ABEND. You can use an abnormal termination exit routine to handle program interruptions, but the interface is more complex than for a program interruption exit (see Section 41.8 on page 976).

The basic mechanism used by the CPU for handling interruptions is illustrated in Figure 770 on page 973 (also shown in Figure 16 on page 55).

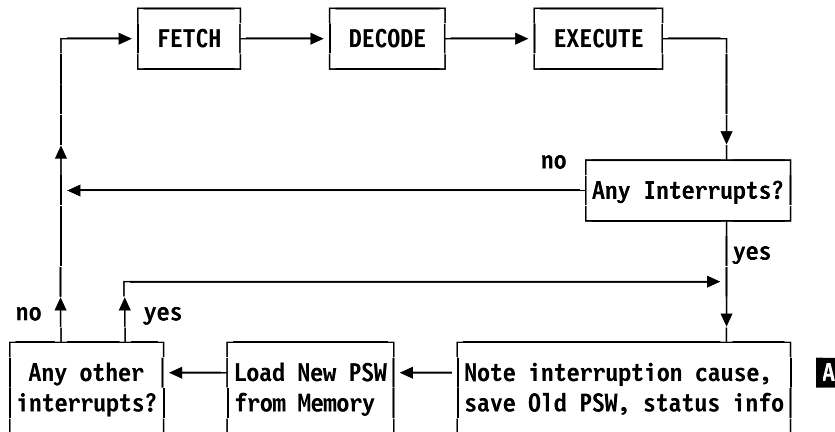


Figure 770. Instruction cycle with interruptions

After determining at **A** that the interruption can be processed, the system analyzes the cause and checks whether an exit routine has been established (by the interrupted program) for that type of interruption.

41.7.1. Program Interruptions

If a program interruption exit has been established, the system saves relevant status information in a work area called a "Program Interruption Element" or "PIE", and the exit routine is given control with a pointer to the work area.

Your program can receive control when any of the 15 program interruptions occurs if the exit routine has requested control. The exit can take corrective action if it chooses.

You can also establish multiple exits: each supersedes the previous, and you can terminate them as needed. Only one exit is active at a time.

41.7.2. Establishing a Program Interruption Exit

Use the ESPIE or SPIE[308] macro to establish a Program Interruption Exit. SPIE is for applications using only 24-bit addressing, while ESPIE is for 24- or 31-bit mode applications; we'll use ESPIE for our examples.

The expansion of either macro creates a small "Program Interruption Control Area", or PICA[*] that defines a data structure with the address of the exit routine, a pointer to an argument list to be passed to your exit, and mask bits indicating which of the 15 program interruption types should cause the system to pass control to your exit.

For example, to request that your exit routine named **ProgInt** be given control for any of the 15 possible program interruption types, you can write

```
         ESPIE SET,ProgInt,((1,15))
```

Figure 771. Establishing a program interruption exit

where the first operand SET establishes the exit, the second operand is the name of your exit routine, and the third operand specifies interruption types 1 through 15. The expansion of this macro is shown in Figure 772 on page 974, where the PICA is in statment numbers 16-20.

[308] The macro names can be understood to mean "(Extended) Set Program Interruption Exit".

[*] This PICA is unrelated to typography or cravings for unnatural foods.

```
                                     11         ESPIE SET,ProgInt,((1,15))
                                     12+*       MACDATE = 08/15/81
00000C                               13+        CNOP  0,4
00000C 4D10 C020           00020     14+        BAS   1,*+20
                      00010          15+IHB00002 EQU   *
000010 00000038                      16+        DC    A(ProgInt)         USER EXIT ROUTINE ADDRESS
000014 00000000                      17+        DC    A(0)               USER PARAMETER LIST ADDRESS
000018 7FFF                          18+        DC    B'0111111111111111' INTERRUPTION MASK
00001A 0000                          19+        DC    B'0000000000000000'
00001C 00000000                      20+        DC    A(0)               RESERVED
000020 4100 0004           00004     21+        LA    0,4                SET FUNCTION CODE
000024 41F0 001C           0001C     22+        LA    15,28              SVC ROUTER CODE
000028 0A6D                          23+        SVC   109
00002A 5010 C09C           0009C     24         ST    1,MyToken          Save token
```

Figure 772. Expansion of an ESPIE macro establishing a program interruption exit

The 2-byte bit string in statement 18 contains a 1-bit for each type of program interruption you want to examine; in this case, all 15.

When the ESPIE SET macro is executed, the system first creates an "Extended Program Interruption Element" (EPIE) that will hold data about an interruption when it occurs, and then returns a 4-byte "token" in GR1 for you to save.

Maskable Interruptions

If your program has set the Program Mask to disable any of the four maskable interruptions — fixed-point overflow (8), decimal overflow (9), HFP exponent underflow (13), or HFP lost significance (15) — specifying any of those exceptions in a SPIE or ESPIE macro will *re-enable* that exception.

For more information about the maskable exceptions, see the initial description at " 4.6. Exceptions and Program Interruptions (*)" on page 56 and the examples of the SPM instruction at " 16.9. Retrieving and Setting the Program Mask (*)" on page 234.

41.7.3. Terminating a Program Interruption Exit

The token returned to you when you established an exit identifies the PICA generated by the *previous* ESPIE macro (if any); you use it when you want to cancel the current program interruption exit and restore control to any previous exit.

You can consider program interruption exits as a stack of programs: the most recent is on top of the stack. Then, if another module in your program establishes its own program interruption exit, when that second exit is canceled, the most previously established exit resumes effect.

Note that if the active exit does not process a given type of program interruption, control is not passed "down the stack" to an earlier exit; the program will be ABENDed by the system.

To terminate the current program interruption exit, issue an ESPIE macro with the RESET operand, and the token that you received when you established the exit:

```
         ESPIE RESET,MyToken      End current exit
```

Figure 773. Terminating a program interruption exit

The expansion of this macro is shown in Figure 774:

```
                                    770         ESPIE RESET,MyToken
                                    771+*       MACDATE = 08/15/81
000052 5810 C0AC           000AC    772+        L     1,MyToken         GET TOKEN
000056 4100 0008           00008    773+        LA    0,8               RESET FUNCTION CODE
00005A 41F0 001C           0001C    774+        LA    15,28             SVC ROUTER CODE
00005E 0A6D                         775+        SVC   109
```

Figure 774. Expansion of an ESPIE macro terminating a program interruption exit

If you want to cancel *all* program interruption exits, specify a zero token:

```
         ESPIE RESET,0           Cancel all exits
```

41.7.4. Handling a Program Interruption

When a program interruption occurs and the currently active PICA requests control, the system places information in the EPIE:

- the general registers in effect at the point of interruption
- the 8-byte ESA/390-mode old PSW (see Figure 775), containing
  - the address and addressing mode of the instruction *following* the interrupted instruction (IA and A in Figure 775)
  - the condition code, program mask (see CC and PM in Figure 775) and the instruction length code
  - the interruption code and Data Exception Code; if the DXC field is zero, the exception was due to invalid decimal data; otherwise, see Section 34.4.1. on page 649 for a description of possible floating-point exceptions

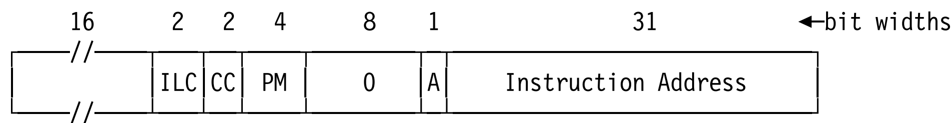


Figure 775. ESA/390-mode old PSW in EPIE

The most useful contents of the EPIE are sketched in Table 437. (The full EPIE is described in the *MVS Data Areas* manual referenced in the Bibliography on page 1057.)

Offset (dec)	Offset (hex)	Length	Type	Description
0	0	4	CL4	Control block identifier C'EPIE'
4	4	4	A	Parameter list address
8	8	64	XL64	32-bit GPRs at time of interruption
72	48	8	XL8	8-byte ESA/390 old PSW at time of interruption
74	4A	1	BL1	Condition Code and Program Mask
76	4C	4	A	AMODE bit and 31-bit address of next instruction to execute
81	51	1	BL1	Instruction Length code (bit positions X'06')
83	53	1	XL1	Interruption code
87	57	1	BL1	Data Exception code (if Interruption code is 7)
153	99	1	BL1	Set bit X'40' to resume execution by restoring the 64-bit GPRs rather than the 32-bit GPRs
160	A0	128	XL128	64-bit GPRs at time of interruption
288	120	8	XL8	Breaking Event Address
296	128	16	XL16	16-byte old PSW at time of interruption

Table 437. Partial contents of Extended Program Interruption Element (EPIE)

Using the information in the EPIE, your Program Interruption Exit can do many things, such as:

1. Examine the interruption condition and do nothing; simply return to the address in GR14 on entry to the exit routine.
2. Print an error message and resume execution at the next instruction.
3. If the interruption was due to invalid data, analyze the instruction to determine where the data is; then try correcting the data or the register pointing to it. Then use the ILC in the EPIE to subtract the length of the interrupted instruction from the old PSW, and try again

(being sure to remember that this is a second try, so you don't create an interruption loop!).[309]

4. Modify the old PSW address in the EPIE so that control will resume at a different place in your program after your exit returns via the system.
5. If your exit determines that continuing execution is a bad idea, set bit '10' at offset 153 (X'99') and return to the system; it will automatically issue an ABEND. (Don't modify anything else in the EPIE.)

Many other possibilities will come to mind.

Be Careful!

1. The linkage from the system to your program interruption exit is *not* standard: do not try to save any registers in the “caller's” save area.
2. If you do I/O from your exit routine, you must establish a local save area pointed to by GR13. Because they are used by I/O routines, you must first preserve GR14, GR15, GR0, and GR1.
3. Your exit routine *must* return to the interrupted program via the return address in GR14 when the exit was entered.

Exercises

41.7.1.(2))+ Write short examples of instruction sequences that will cause each of the 15 types of program interruption. (You will want to use this information when you solve Programming Problem 41.6.)

41.8. Abnormal Terminations of Any Kind

Note!

This topic is quite complicated; this section can at best give a brief outline of its key components and actions. For a detailed overview, see the topic “Providing Recovery” in the *MVS Assembler Services Guide*, referenced in the Bibliography on page 1057.

As when you establish an exit to handle program interruptions with ESPIE, you can specify a routine to be given control on *any* abnormal termination condition. That routine (called a *recovery* routine) can capture information about the cause and location of the problem, maybe take remedial action to repair or bypass the error and resume execution at a chosen point in your program, or release any resources the program may have acquired and terminate a bit more “gracefully”.

The overall flow of control can be visualized in simplified form in Figure 776 on page 977.

[309] On a very early System/360 Model 67, one particular program interruption at an address at a specific offset from a doubleword boundary would resume execution not at the next sequential instruction, but at the previous doubleword boundary. If that was the address of the interrupted instruction, an interruption loop would occur, through no fault of the programmer. It was quickly corrected.

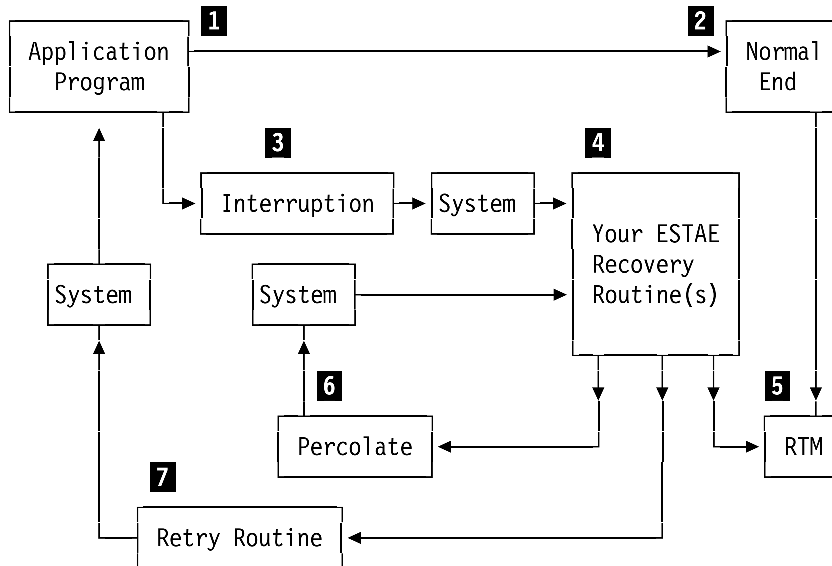


Figure 776. Sketch of interruption handling control flow

In this basic scenario, you are responsible for writing three routines: the Application Program **1**, the ESTAE recovery routine(s) (ESTAE exit(s)) **4**, and the Retry Routine **7**. These can be separate programs, or parts of the Application Program.

Under normal conditions, you expect the Application Program to complete with a Normal End **2**, and control passes to the "Recovery/Termination Manager" (RTM) to do any necessary final actions. If an Interruption **3** occurs, or if your Application Program **1** issues an ABEND macro, the System will give control to the most recent of your Recovery Routines **4**, and provide useful information in a "System Diagnostic Work Area" (SDWA). If no Recovery Routine is available, control will pass directly to the Recovery/Termination Manager (RTM) **5**, which will do its best to clean up any resources held by your program and generate whatever diagnostic information it can.

Assuming a Recovery Routine is available, you can make several choices:

1. You can decide that the problem cannot readily be fixed, generate as much diagnostic information as you can, and pass control to the RTM **5** to terminate execution (possibly with additional diagnostic information such as a "memory dump").
2. You can choose to do some repairs and pass control to the Recovery/Termination Manager **5**, or you can decide that the problem may be identifiable by the next of the Recovery Routines on the "exit stack", and indicate to the System that recovery control should "percolate" **6** to the next exit routine.
3. You can process the interruption condition so that it is partly or completely repaired, and decide to pass control back to the application program by way of the Retry Routine **7**; it can complete fixing the problem, or completely bypass it.

Both percolation and retry require the SETRP macro, which we'll review on page 979.

The most general form of exception handling is provided by the STAE, ESTAE, and ESTAEX macros,[310] all of which support Standard, List, and Execute forms.

STAE was the earliest form, and executes only in 24-bit addressing mode. ESTAE was introduced next, and executes in both 24- and 31-bit modes. ESTAEX is the most general form, and can execute in all three addresing modes. (See the details in the *Assembler Services Reference* manual listed in the Bibliography.) The following discussion will use ESTAE; ESTAEX is very similar.

310 The names are generally understood to mean "(Extended) Specify Task Abnormal Exit".

41.8.1. The ESTAE Macro

Issuing ESTAE establishes an exit routine that will be given control on an abnormal termination condition in your program. If you issue multiple ESTAEs, you can either replace the current exit with a new one (like ESPIE), or (unlike ESPIE) you can form a "chain" or "stack" of exit routines, each of which can decide to take corrective action, or to pass the problem to the previously defined exit routine. This passing of control is called "percolation".

Because program interruptions can cause an abnormal termination, an ESTAE exit can also process program interruption conditions, but the ESTAE interface is both more powerful and much more complex than ESPIE's.

The ESTAE macro has this general form:

```
ESTAE  two_positional_operands,multiple_keyword_operands
```

These are the most important operands for most programs; default values are underlined.

exitname This is the first positional operand, giving the entry point name of the recovery routine to be placed at the top of the exit "stack".

If you specify **0** for this operand, the name of the current exit routine (at the top of the stack) is removed.

CT|OV This second positional argument is either **CT** or **OV**. Specifying **OV** means that the current exit routine will be replaced ("overlaid") by the new one provided in the first operand. Specifying **CT** means that the new exit routine will be added ("concatenated") to the top of the stack of exit program names. It will be entered first by the operating system when an interruption occurs, but it can "percolate" control to the previously defined exit routine.

PARAM= This keyword operand specifies the address of a "parameter area" created by the Application Program that can provide useful information for the Recovery Routine, and where that routine can place data that may be useful for the application.

PURGE= This keyword operand specifies the actions you want to take with currently active I/O operations.

- **NONE** Don't do anything; let the operations continue. (This means that any interruptions caused by the active I/O could cause re-entry to your exit routine.)
- **QUIESCE** Save the status of *pending* I/O requests so they can be restarted if desired. Currently active I/O operations will be purged.
- **HALT** Don't save pending requests. Currently active I/O operations will be purged.

ASYNCH= This keyword operand specifies whether certain types of interruption should be allowed while your recovery routine is in control:

- **NO** No other interruptions should be allowed.
- **YES** Must be specified if your exit routine will request system services that could generate interruptions, or if you specified **PURGE=QUIESCE** or **PURGE=NONE** and interruptions are required to complete processing of certain I/O activities.

SDWALOC31= You can specify whether the "System Diagnostic Work Area" (SDWA) created by the System should be located in 24- or 31-bit addressable storage:

- **NO** The SDWA will be in 24-bit storage.
- **YES** **YES** is required if the application program is running in 31-bit addressing mode, or if it is using 64-bit general registers.

The satisfactory-completion return codes (in GR15) from ESTAE are 0 and 4 (you specified OV but there was no previous exit to overlay, so the System treated your request as though you had specified CT). All other return codes indicate an error.

A simple example of an ESTAE macro is shown in Figure 777.

```
                               4          ESTAE  EXIT,PARAM=PLIST
000000                         7+         CNOP  0,4                    ESTAB. FULL WD. BOUND. ALIGN.
000000 A715 000E      0001C    8+         BRAS  1,*+28                 LIST ADDR IN REG1 SKIP LIST
000004 16                      9+         DC    AL1(22)                FLAGS FOR TCB, PURGE,          X
                                +                                      ASYNCH AND CANCEL
000005 000000                 10+         DC    AL3(0)                 FIELD NO LONGER USED
000008 00000084               11+         DC    A(PLIST)               STAE EXIT PARM. LIST ADDR.
00000C 00000000               12+         DC    A(0)                   SPACE FOR TCB ADDR
000010 00                     13+         DC    AL1(0)                 FLAGS FOR TERM,RECORD,SDWALOC31
000011 01                     14+         DC    AL1(1)                 THIRD FLAG BYTE
000012 0000                   15+         DC    AL2(0)                 RESERVED
000014 00000000               16+         DC    A(0)                   SPACE FOR TOKEN
000018 00000080               17+         DC    AL4(EXIT)              FOUR BYTE EXIT ADDR
00001C 4100 0100      00100   18+         LA    0,256(0,0)             CREATE & PARMLST EQ 0
000020 4110 1000      00000   19+         LA    1,0(0,1)               MAKE REG1 POS.  XCTL=NO
000024 0A3C                   20+         SVC   60                     ISSUE STAE SVC
```

Figure 777. A simple ESTAE macro.

You can see the references to the exit and parameter-area addresses; the other fields are for arguments we haven't discussed.

41.8.2. Interruption Processing

Your ESTAE exit receives control from the System in the addressing mode that was in effect when the ESTAE macro was issued, *not* in the current mode. You should do several things first:

1. Save the return address in GR14, so you can return control to the System. GR2 contains the address of the “parameter area” that you specified on the **PARAM=** operand of the ESTAE macro. (The parameter area address is also in the SDWA field SDWAPARM.)
2. Test GR0:
   - If c(GR0)=12 (X'0C'), no SDWA has been provided by the System, and GR13 does not point to a save area. Your options are quite limited, and it's best to terminate the program.
   - If c(GR0)≠12, GR1 holds the address of the SDWA, GR13 points to a standard save area, and the address of the parameter area is also in the SDWAPARM field of the SDWA. (The names of the SDWA fields are defined in the DSECT generated by the IHASDWA macro, and described in the *MVS Data Areas* manual listed in the Bibliography.)
     - The completion code (also known as the ABEND code) is in the SDWA's SDWACMPC field, and if the SDWARCF bit is 1, the SDWACRC contains the reason code (that you could have provided on an ABEND macro).
     - Check the SDWAPERC bit: if it's 0, this is the first recovery routine, and there has been no previous percolation.

The information provided by the System in the SDWA to the exit routine is very rich, including the contents of the general registers, the 8-byte PSW in effect at the time of the original ABEND, and other useful data. The SDWA is described in the *MVS Data Areas* manual; see the Bibliography for details.

Your recovery routine may need to communicate with the application program. The best way to do this is to use the parameter area you specified on the PARAM= operand of the ESTAE macro. This area can contain addresses of parts of the application program and of acquired storage, addresses that may be needed to locate data or that may be needed by your retry routine.

41.8.3. Percolation and Retry

If your exit wants to percolate to another exit, or retry execution at a point in the application, it must issue the SETRP (Set Return Parameters) macro, described in the *Assembler Services Reference* manual. Depending on the arguments on the SETRP macro when you have used it to return control to the System, the System will percolate to the next exit routine on the exit “stack”, or give control to your retry routine.

Note that if the SWDACLUP bit in the SDWA is 1, retry is not allowed.

The general form of the macro is:

```
SETRP keyword_arguments
```

You should use SETRP only if the System provided a SDWA to your exit routine. If there is no SDWA, it's best not to retry.

These are the most important operands for most programmers; default values are underlined.

RC= <u>0</u>|4. This argument specifies whether you want to percolate (0) or retry (4).

REGS= (reg1,reg2). The register or range of registers to be restored from the standard save area addressed by GR13. Specify the register values as you would for a Load Multiple instruction: (reg1,reg2). The macro will generate a BR 14 instruction to return control to the System. (If you omit this argument, you must create your own branch instruction to return control to the System.)

The following arguments may be specified only if you specified **RC=4** (retry).

RETADDR= The address of your retry routine. It can be a separate routine; your exit routine can request that the System resume execution of your program at whatever point you choose.

FRESDWA= **<u>NO</u>**|YES. This argument specifies whether (YES) or not (NO) you want the SDWA to be freed before your retry routine receives control. If you specify NO, the retry routine must free the SDWA, using length and subpool information in the SDWA.

There are many other macro operands not mentioned here; see the *MVS Assembler Services Reference* in the Bibliography for details.

Percolation: If you chose to percolate (RC=0) it's best to repair as much of the problem as possible, such as releasing unneeded resources, depending on the type of interruption.

Before you issue the SETRP macro to return control to the System (and then to the next recovery routine on the "exit stack"), check whether a SDWA was provided: if not, set c(GR15) to zero. If yes, you could update the SDWA with data that might be useful if and when the program continues, or is terminated.

When the SERTP macro passes control to the system, the system removes the current ESTAE exit from the top of the exit stack; if the stack is now empty, the system will abnormally terminate your program and control will pass directly to the RTM.

Retry: If you choose to retry (RC=4), you are passing control back to the Application Program's retry routine (which can be an existing module or part of the application). The current ESTAE exit on the top of the exit "stack" is *not* freed. If your retry routine creates another interruption, control will pass to the same recovery routine that just gave control back to the application via the retry routine. Be careful: you could cause an error-processing loop.

41.8.4. Summary

This topic is far more extensive than this overview can properly describe. The cited references provide all the information you may need.

From *your* perspective (*not* the System's!) you can think of your program as being in one of several states:

1. Executing; it is doing what it was intended to do.
2. In your ESTAE-exit recovery routine, your program having created an interruption condition.
3. "Abnormal end": no recovery routines, or your recovery routine(s) failed.
4. Terminated.

From the *System's* perspective, your program can be in several states:

1. Executing; doing something.
2. Having executed an ESTAE macro, which creates an exit that can then be in one of four states:

a. Defined and activated: known to the System; you provided an exit name on ESTAE, and available to receive control if an interruption occurs.

b. In control; the System has passed control to the exit.

c. No longer in control, because it has either percolated or retried.

d. Deactivated and undefined: the System will no longer pass control to it, and not known to the System.

3. Terminating abnormally: you provided no exit routines, or your exit routines failed.
4. Terminating normally.

41.9. Summary

Instructions Discussed in this Section

The instruction mnemonics and opcodes are shown in the following table:

Mnemonic	Opcode
PC	B218

Mnemonic	Opcode
SVC	0A

The instruction opcodes and mnemonics are shown in the following table:

Opcode	Mnemonic
0A	SVC

Opcode	Mnemonic
B218	PC

Terms and Definitions

ABEND
Abnormal termination (**AB**normal **END**) of a program prior to its expected completion, either requested by the executing program or imposed by the operating system.

abnormal termination
The termination of a program prior to its expected completion. It can be requested by the executing program, or imposed by the operating system.

keyword argument
An argument to a macro definition where the keyword name supplied by the macro definition is followed by an equal sign and an optional value.

positional argument
An argument to a macro definition determined by counting from the first operand, ignoring any keyword arguments. The contents of the name field of the macro invocation can be treated as a positional argument by macro definitions.

program interuption
An interruption that is caused by the program that can be fixed by source statement modifications. Distinct from a system interruption generated by the operating system.

recovery routine
A program written in the hope that your program can correct whatever condition caused the operating system to terminate the program.

system interruption
An interruption caused by an improper interaction between the executing program and a system service.

system service
A function of by the operating system, providing capabilities not allowed to most problem state programs.

Programming Problems

Programming Problem 41.1.(1) Revise the little program in Figure 33 on page 81 to use system macros rather than PRINTOUT.

Programming Problem 41.2.(3)+ Write a program to read 80-byte fixed length records, and write them to a 121-byte print data set (with carriage-control characters), preceded by a sequence number giving the number of the input record. Verify that each DCB opened correctly, and take appropriate actions to notify the user if not.

For sample input records, you can use the data from Programming Problems 24.13 or 24.14. on page 400.

Programming Problem 41.3.(3)+ Write a program with a program interruption exit that generates each the 15 possible interruption types in turn. For each interruption, generate a message describing the interruption type and the address of the interrupted instruction. Then, return to the mainline program to generate the next interruption.

Programming Problem 41.4.(1)+ Write a program using ESPIE to establish a program interruption exit, and display the token returned in GR1. Then establish a second exit, and display the token returned in GR1. Can you determine what it might refer to? Then, use the RESET option of ESPIE to terminate both exits.

Programming Problem 41.5.(3)+ Write a short program that issues an ESPIE macro that designates an exit routine in your program. Then, generate a program interruption that causes entry to your ESPIE exit routine. The exit should display (print) information about the interruption:

- the interruption code
- the address of the failed instruction
- the failed instruction
- any relevant register contents
- any relevant storage fields
- anything else that might be helpful

Then, return to the main program following the failed instruction, to complete execution normally.

Programming Problem 41.6.(4)+ Using your solution to Programming Problem 41.5 as a starting point, write a program to generate as many of the 15 program interruptions as you can, displaying the same information as before, plus any additional information you believe may be useful. (For example, for a decimal data interruption, show the invalid operands; this may require calculating some Effective Addresses.) (Your solution to Exercise 41.7.1 will be useful.)

Programming Problem 41.7.(3)+ Write a program with an ESTAE exit. Then, generate a program interruption as in Programming Problems 41.5 or 41.6. Your recovery routine should provide at least the same information as in Problem 41.5.

Programming Problem 41.8.(4) Write a program with an ESTAE exit, and generate an interruption condition *other than* a program interruption, and provide at least the same information as in Problem 41.5.

42. Reenterability and Recursion

```
      444       2222222222
     4444     222222222222
    44 44     22         22
   44  44                22
  44   44                22
 44444444444             22
444444444444           22
       44           22
       44         22
       44       22
       44     222222222222
       44     222222222222
```

42.1. Reenterability

We'll start by outlining what "reenterability" means technically, then what it implies in practice.

42.1.1. What it Means in General

There are many ways to understand reenterability:

1. A program does not modify itself.
2. A program does not modify itself and is loaded into memory-protected storage.
3. A program that produces correct results when multiple units of work execute it concurrently. It can modify itself if none of the units of work rely on or can detect the modification. For example:

   ```
   L    0,X     or  IC   0,*      or  OI   *,0      or  NI   *,X'FF'
   ST   0,X         STC  0,*-4
   ```

   Such a program modifies itself, but the modification is not detectable unless the program has been loaded into protected storage.
4. Many programmers use the term "reentrant" to describe a reenterable program.[311] Mathematically, "reentrant" describes a closed curve with at least one concave segment.

Sometimes people think reenterability also means recursion; they're quite distinct. We'll discuss recursion in Section 42.2 on page 987.

[311] It has been claimed that "reenterable" was the original term, starting around 1964. Some years later someone apparently decided that word looked too complex, and substituted "reentrant", without knowing its mathematical definition.

42.1.2. What it Means in Practice

Before starting, you should evaluate where, how, and when the module will be used, to decide whether or not it should be reenterable. But note that if you write all programs and subprograms to be reenterable, you can use their components more easily in other applications, whether or not the complete application is reenterable.

42.1.3. Assembly-Time Considerations

1. Specifying the RENT option asks the Assembler to check for obvious cases of self-modification, but there are many ways for a program to modify itself that the Assembler cannot detect. Specifying a RSECT control section does the same checking on a per-section basis,[312] but neither will detect d(b) references like this:

```
         LA    1,*
         STC   1,0(1)           Not detected by RENT or RSECT!

         STC   2,*              Detected by RENT and RSECT!
```

2. Anything the routine may modify should be in acquired storage, or in a work area provided by the caller.
3. These practices can make it easier to write a reenterable program (see Figure 778 on page 985):
   a. Define a DSECT for all local variables (but not constants). It should contain a local save area for calls to other external routines or to system services.
   b. On completion of entry linkage, allocate storage for the local variables, and assign a base register for this storage area.
   c. Initialize local variables as required.
   d. Release the local storage as part of the exit linkage.
4. Use the List and Execute forms of most macros, unless you have verified that the Standard form does not generate inline store instructions. See Figures 748, 750, 752, and 754 starting on page 959 for examples of macros that can be used in reenterable programs without needing to use the List and Execute forms.

42.1.4. At Linking Time

When you link a reenterable program using the z/OS Program Management Binder, you must specify either the RENT or REFR option; these options describe your program's property (see the Binder manuals shown in the Bibliography for details):

NONE The module cannot be reused. A new copy must be brought into memory for each use.

SERIAL The module is serially reusable. It can only be executed by one task at a time; when one task has finished executing it another task can begin. A serially reusable module can modify its own code, but when it is re-executed it must initialize itself or restore any instructions or data that have been altered. (Sometimes known as **REUS**.)

RENT The module is reenterable. It can be executed by more than one task at a time. A task can begin executing it before a previous task has completed execution. A reenterable module cannot modify its own code. In some cases, the operating system may load a reentrant program into an area of virtual storage that cannot be modified. Reenterable modules are also serially reusable.

REFR The module is refreshable. It can be replaced by a new copy during execution without changing the sequence or results of processing. A refreshable module cannot be modified during execution. Refreshable modules are also reenterable and serially reusable. A module can be refreshable only if all the control sections within it are refreshable.

---

[312] Except for enabling checks for simple cases of self-modification within its section, RSECT is the same as CSECT.

The Binder (like the Assembler) trusts your claims about your program's reenterability, and does not try to verify them.

42.1.5. Techniques

The following examples show how you can write a reenterable program that runs in 24- or 31-bit mode and resides below the 16MB "line".

At the start of a reenterable program, you normally acquire storage for save areas, working data, macro lists, and so on. A partial sketch of the entry and exit linkages for a program executing in 24- or 31-bit addressing mode and residing below the 16MB "line" is shown in Figure 778; more will be added for a functioning program.

```
RentProg RSect ,
         SAVE  (14,12),,*          Save caller's registers
         LR    12,15               Copy base register
         Using RentProg,12         Establish addressability
         GETMAIN R,LV=WorkLen      Get working storage
         ST    13,4(,1)            Save back chain
         ST    1,8(,13)            Store caller's forward chain
         LR    13,1                Point R13 to the work area
         Using WorkArea,13         Addressability for work area
         XC    8(4,13),8(13)       Clear our forward chain
         - - -                     Do whatever needs doing
         L     13,Save+4           Reload address of caller's area
         LR    1,13                Put work area address in GR1
         FREEMAIN R,A=(1),LV=WorkLen   Free the acquired storage
         RETURN (14,12)            Return to caller or system
WorkArea DSect ,
Save     DS    9D                  Local save area
Buffer   DS    CL80
OpenList OPEN  (,,,),MF=L          List to OPEN two DCBs
CLoseLst CLOSE (,,,),MF=L          List to CLOSE two DCBs
         - - -
         DS    0D                  Align to doubleword
WorkLen  Equ   *-WorkArea          Length of the work area
```

Figure 778. Skeleton form of a reenterable program

Figure 778 shows how to allocate working storage, set up a save area, then to free it and return. Now, we'll add some "working" statements.

Suppose this little program reads records from INDCB, processes them, and writes them to OUTDCB. In Figure 778 we omitted the DCB macros, so we'll add those and the GET and PUT macros in Figure 779 on page 986, assuming that the "housekeeping" instructions are unchanged. First, because the DCBs are modified during execution, they must be copied to the work area.

```
        - - -
        MVC   WDCB1(WDCB1L),INDCB     Copy INDCB to the work area
        MVC   WDCB2(WDCB2L),OUTDCB    Copy OUTDCB to the work area
        LA    2,WDCB1            Point to working input DCB
        LA    3,WDCB2            Point to working output DCB
        OPEN  ((2),INPUT,(3),OUTPUT),MF=(E,OpenList) Open both
GetRec  GET   (2),Buffer         Read a record
        - - -                    Process it
        PUT   (3),Buffer         Write it out
        J     GetRec
EndFile CLOSE ((2),,(3),)        Close the DCBs
        - - -
INDCB   DCB   DDName=INFILE,LRECL=80,BLKSIZE=8000,MACRF=GM,DSORG=PS,  X
              EODAD=EndFile
INDCBL  Equ   *-INDCB            Length of Input DCB
OUTDCB  DCB   DDName=OUTFILE,LRECL=80,BLKSIZE=8000,MACRF=PM,DSORG=PS
OUTDCBL Equ   *-OUTDCB           Length of Output DCB
        - - -
WorkArea DSect ,
        - - -
WDCB1   DCB   DSORG=PS,MACRF=GM,DDNAME=X   Working DCB1
WDCB1L  Equ   *-WDCB1            Length of Working DCB1
WDCB2   DCB   DSORG=PS,MACRF=PM,DDNAME=X   Working DCB2
WDCB2L  Equ   *-WDCB2            Length of Working DCB2
```

Figure 779. I/O macros in a reenterable program

The use of unusual (and identical) DDNames on the working DCBs doesn't matter because they are overwritten after the DCB skeletons are moved to the work area. However, you should always check your assembly listing to verify that the length of each DCB matches the length of the work area to which it's moved. (Here, they should both have length X'60'. This is easily checked by comparing their EQUated lengths.)

The complete assembled program is shown in Figure 780.

```
                                    1          Print NoGen
000000                00000 00188   2 RentProg RSect ,
000000 47F0 F00E            0000E   3          SAVE  (14,12),,*        Save caller's registers
000012 18CF                         8          LR    12,15             Copy base register
                R:C  00000          9          Using RentProg,12       Establish addressability
000014 4510 C01C            0001C  10          GETMAIN R,LV=WorkLen    Get working storage
000022 50D0 1004            00004  16          ST    13,4(,1)          Save back chain
000026 5010 D008            00008  17          ST    1,8(,13)          Store caller's forward chain
00002A 18D1                        18          LR    13,1              Point R13 to the work area
                R:D  00000         19          Using WorkArea,13       Addressability for work area
00002C D703 D008 D008 00008 00008  20          XC    8(4,13),8(13)     Clear our forward chain
000032 D25F D0A8 C0C8 000A8 000C8  21          MVC   WDCB1(WDCB1L),INDCB     Copy INDCB to the work area
000038 D25F D108 C128 00108 00128  22          MVC   WDCB2(WDCB2L),OUTDCB    Copy OUTDCB to the work area
00003E 4120 D0A8            000A8  23          LA    2,WDCB1           Point to working input DCB
000042 4130 D108            00108  24          LA    3,WDCB2           Point to working output DCB
```

Figure 780 (Part 1 of 2). Assembly listing for a simple reenterable program

```
000046 4110 D098           00098    25          OPEN  ((2),INPUT,(3),OUTPUT),MF=(E,OpenList) Open both
000070 1812                         38 GetRec   GET   (2),Buffer         Read a record
                                    44 *        - - -                    Process it
00007C 1813                         45          PUT   (3),Buffer         Write it out
000088 A7F4 FFF4           00070    51          J     GetRec
00008C 4110 D0A0           000A0    52 EndFile  CLOSE ((2),,(3),),MF=(E,CloseLst)  Close both DCBs
0000AE 58D0 D004           00004    63          L     13,Save+4          Reload address of caller's area
0000B2 181D                         64          LR    1,13               Put work area address in GR1
0000B4 47F0 C0BC           000BC    65          FREEMAIN R,A=(1),LV=WorkLen   Free the acquired storage
0000C2 98EC D00C           0000C    71          RETURN (14,12)           Return to caller or system
                                    74 INDCB    DCB   DDName=INFILE,LRECL=80,BLKSIZE=8000,MACRF=GM,DSORG=PS,  X
0000C8 0000000000000000                               EODAD=EndFile
                           00060   128 INDCBL   Equ   *-INDCB            Length of Input DCB
000128 0000000000000000            129 OUTDCB   DCB   DDName=OUTFILE,LRECL=80,BLKSIZE=8000,MACRF=PM,DSORG=PS
                           00060   128 OUTDCBL  Equ   *-OUTDCB           Length of Output DCB
000000               00000 00168   184 WorkArea DSect ,
000000                             185 Save     DS    9D                 Local save area
000048                             186 Buffer   DS    CL80
000098 00                          187 OpenList OPEN  (,,,),MF=L         List to OPEN two DCBs
0000A0 00                          193 CLoseLst CLOSE (,,,),MF=L         List to CLOSE two DCBs
0000A8 0000000000000000            199 WDCB1    DCB   DSORG=PS,MACRF=GM,DDNAME=X
                     00060         254 WDCB1L   Equ   *-WDCB1            Length of Working DCB1
000108 0000000000000000            255 WDCB2    DCB   DSORG=PS,MACRF=GM,DDNAME=X
                     00060         310 WDCB2L   Equ   *-WDCB2            Length of Working DCB2
000168                             311          DS    0D                 Align to doubleword
                     00168         312 WorkLen  Equ   *-WorkArea         Length of the work area
                                   313          End
```

Figure 780 (Part 2 of 2). Assembly listing for a simple reenterable program

Exercises

42.1.1.(3) **Note**: This exercise is intended as a mental exercise in violating reenterability, *not* to illustrate a useful or proper programming technique!

Write a program with a loop, in which a single instruction modifies itself on each iteration through the loop, in such a way that the instruction is different each time. That is, an instruction like `MVI  *+1,0` does not change on each iteration. Display enough of the program's behavior to convince yourself that your solution works.

42.2. Recursion

Recursion appears in varied forms in many programming languages. For example, the common practice of grouping nested mathematical expressions in parentheses often implies the need for a recursive routine to parse the complete expression. In Section 8.6 on page 103, the Assembler Language defines an operand as formed from one to three expressions. Each expression is defined as a “`factor`” or as “`±factor`” or as “`factor±factor`”; a factor is defined in turn as a “`primary`” or as a “`primary*primary`” or as a “`primary/primary`”. Finally, a primary is defined as a “`term`” or as a parenthesized expression, “`( expression )`”. It is this last item that makes the definition of an expression recursive; you could be scanning an expression and encounter an opening parenthesis and find that you must (recursively) scan a new expression that is part of the expression you were previously scanning.

As another example, suppose the Assembler Language supported some types of nested literals, as in “`=A(=F'7')`”.[313] After recognizing the “`=A(`” portion of the outer literal, the Assembler would normally parse an address-constant operand, but in this case the “`=F'7'`” operand would require a recursive re-entry to the literal-operand scanner.

Mathematically, a recursive function is one that is defined in terms of itself. In programs, a recursive routine calls itself, either directly or indirectly. An example of a routine that can be imple-

[313] At the time of this writing, HLASM doesn't support nested literals; but you can always ask.

mented both with or without recursion is the familiar Factorial function; we saw iterative solutions in Programming Problems 18.5, 33.2, and 36.4. Its recursive definition is

1. Factorial: for integers N ≥ 0.

```
Fact(N) = 1 if N = 0 or 1
Fact(N) = N × Fact(N-1) otherwise
```

Some other functions having recursive definitions are:

2. Product: for integers M, N ≥ 1.[314]

```
Prod(N,M) = M if N = 1
Prod(N,M) = N if M = 1
Prod(N,M) = N + Prod(N,M-1) otherwise
```

3. Greatest Common Divisor: for integers M, N ≥ 1.[315]

```
GCD(N,M) = N if M = 0
GCD(N,M) = GCD(M,mod(N,M)) otherwise
```

   Note that GCD(M,N) = GCD(N,M). Also, the Least Common Multiple function LCM(N,M) = N×M/GCD(N,M).

   Another interesting GCD result is that $GCD(n^{A}-1,n^{B}-1)=(n^{GCD(A,B)})-1$.

4. Fibonacci numbers: for N ≥ 1.

```
FIBO(N) = 1 if N ≤ 2
FIBO(N) = FIBO(N-1) + FIBO(N-2) otherwise
```

   We've seen many iterative solutions to generating the Fibonacci series; for example, see Programming Problems 16.7-10 and 30.7-8.

5. Remainder: given three positive integers A, b, C, calculate the remainder of A^b/C.

```
REMDR(A^b,C) = REMDR((A^2)^(b/2),C) if A ≤ C and b ≥ 2 is even
REMDR(A^b,C) = mod(A×REMDR((A^2)^((b-1)/2),C),C) if A < C and b ≥ 3 is odd
REMDR(A^b,C) = A if A < C and b = 1
REMDR(A^b,C) = REMDR((mod(A,C))^b,C) if A ≥ C
```

6. Partitions: PART1(N) = number of ways to write N as a sum of positive integers.

```
PART1(N) = 1 if N = 0
PART1(N) = 0 if N < 0
PART1(N) = R(1) - R(2) + R(3) - R(4) + R(5) - ...
where  R(k) = PART1(N-(3×k^2-k)/2) + PART(N-(3×k^2+k)/2)
```

7. Partitions: PART2(N,M) = number of ways to write M as the sum of not more than N positive integers. (Note that PART1(N) = PART2(N,N).)

```
PART2(N,M) = 1 if (a) N = 1 and M ≥ 1, or (b) M = 0, or (c) M = 1
PART2(N,M) = 0 if M < 0
PART2(N,M) = PART2(N-1,M) + PART2(N,M-N)
```

8. Binomial Coefficients: $BINC_{N,M}$ = number of ways to select M objects from a set of N.

```
BINC(N,M) = 1 if M = 0 or M = N
BINC(N,M) = BINC(N-1,M-1) + BINC(N-1,M)
```

   The Binomial coefficients can be displayed in “Pascal's Triangle”: each number is the sum of the two numbers immediately above it to its left and right (using 0 if for the second number for the leftmost and rightmost 1 digits).

[314] Some very early computers evaluated products this way.

[315] Also known as the Greatest Common Factor (GCF) or Highest Common Factor (HCF) function.

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
```

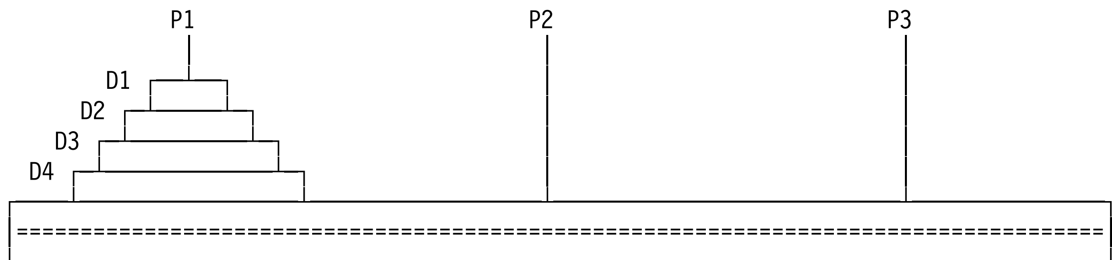
9. Ackermann's Function: for X, Y ≥ 0.

   ```
   ACK(X,Y) = Y + 1 if X = 0
   ACK(X,Y) = ACK(X-1,1) if X > 0 and Y = 0
   ACK(X,Y) = ACK(X-1,ACK(X,Y-1)) otherwise
   ```

   Be careful with the Ackermann function! It grows extremely fast, faster than any exponential function. For example, ACK(3,n) = $2^{(n+3)}-3$, while ACK(4,2) = $2^{65536}-3 \approx 10^{19728}$.

   Note that 2^{65536} = 2**(2**(2**(2**2))), an “exponentiation stack” or “power tower” of five 2's. ACK(4,N) is a “power tower” of (N+3 2's)−3.

10. The “Tower of Hanoi” is a famous problem often involving a recursive solution. suppose you have three pins P1, P2, and P3, and on P1 is a set of disks D1, D2, ... DN in increasing size from top to bottom. For example, with four disks:

    The problem is to move the disks from P1 to P2, subject to these rules:

    a. Only one disk may be moved at a time.
    b. A larger disk is never put on a smaller disk.
    c. At each stage, all disks must be on P1, P2, or P3 (no disks may be held temporarily “elsewhere”).

    A recursive algorithm to accomplish the moves uses the function THMOVE(i,j,M), which moves the topmost M disks from pin P_i to pin P_j, using pin P_k if the topmost disks of pins P_k and P_j are larger than the M-th disk from the top of pin P_i. The steps of the algorithm are:

    Step 1: If M=1, move the topmost disk from P_i to P_j, and return.

    Step 2: Do THMOVE(i,k,M−1) where $i \neq j \neq k$.

    Step 3: Move the topmost disk from P_i to P_j.

    Step 4: Do THMOVE(k,j,M−1) where $i \neq j \neq k$.

    Step 5: Return.

    In the figure above, you would start with THMOVE(1,2,4).

We'll see some of these recursive activities again as Programming Problems.

It is widely believed that a recursive routine must also be reenterable, but this is not so. The recursive routine can maintain its own internal stack; see Programming Problem 42.1 (which is highly recommended).

A common feature of recursive routines is that they must allocate and initialize instances of local variables on each entry, and free them on exit. Many implementations of recursive routines use a stack for local variables, save areas, and so on.

Making a recursive routine also be reenterable can simplify managing local variables if the routine can be loaded into protected storage; any attempt to store into the body of the routine will generate a memory protection error.

Here is an example of a reenterable, recursive routine to evaluate factorials and return the calculated value in GR0. (Remember that the largest argument that can be held in a fullword is 12; trying to evaluate larger values can lead to a variety of unexpected errors.)

```
*        Reenterable, Recursive Factorial Routine
RFact    RSect ,
         STM   14,3,12(13)         Save registers
         LR    3,15                Copy base register
         Using RFAct,3             Provide addressability
         LR    2,1                 Preserve arglist pointer
         LHI   0,LWA               Get length of work area
         GETMAIN R,LV=(0)          Get working storage
         ST    13,4(,1)            Store back chain
         ST    1,8(,13)            Store forward chain
         LR    13,1                Now have local save area
         Using WA,13               Map the work area
         L     2,0(,2)             Get argument address
         L     1,0(,2)             Get argument N
         CHI   1,2                 Is the argument 2?
         JE    Ret                 Fact(2) = 2, no recursion
         JNH   Test                Go test for 0, 1, or < 0
         ST    1,Arg               Save N we're called with
         BCTR  1,0                 Create N-1
         ST    1,NewArg            Store argument for call
         LA    1,NewArg            Create argument address
         ST    1,ArgAddr           Store the address
         LA    1,ArgAddr           Point to argument address
         LARL  15,RFact            Get entry point address
         BASR  14,15               Call ourselves recursively
*        Value returned in GR0
Calc     L     1,Arg               Get the N we were called with
         MR    0,0                 Form N*Factorial(N-1)
Ret      L     2,Save+4            Get caller's save area address
         Drop  13                  No more reference to work area
         ST    1,20(,2)            Store product in R0 slot
         LHI   0,LWA               Set work area length in GR0
         FREEMAIN R,LV=(0),A=(13)  Release our storage
         Drop  3                   No more based references here
         LR    13,2                Restore caller's R13
         LM    14,3,12(2)          Restore registers
         BR    14                  Return to whoever called
Test     LTR   1,1                 Is argument 0 or 1 or < 0
         JM    SetZ                If < 0, return 0
         LHI   1,1                 If 0 or 1, return 1
         J     Ret                 And complete return sequence
SetZ     XR    1,1                 Return 0 for N < 0
         J     Ret                 And return
```

Figure 781 (Part 1 of 2). Example of a reenterable, recursive routine

```
WA       DSect ,                 Work area mapping
Save     DS    5D                Local save area, GR14-GR3
Arg      DS    F                 Value we were called with
NewArg   DS    F                 Argument for recursive call
ArgAddr  DS    A                 Address of the argument
         DS    0D                Round to doubleword
LWA      Equ   *-WA              Length of work area
         End
```

Figure 781 (Part 2 of 2). Example of a reenterable, recursive routine

The assembly listing is shown in Figure 782:

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement
                                      1 *        Reenterable, Recursive Factorial Routine
000000                00000 00082     2 RFact    RSect ,
000000 90E3 D00C            0000C     3          STM   14,3,12(13)       Save registers
000004 183F                           4          LR    3,15              Copy base register
                R:3  00000            5          Using RFAct,3           Provide addressability
000006 1821                           6          LR    2,1               Preserve arglist pointer
000008 A708 0038            00038     7          LHI   0,LWA             Get length of work area
                                      8          GETMAIN R,LV=(0)        Get working storage
00000C 4510 3010            00010     9+         BAL   1,*+4                  INDICATE GETMAIN
000010 0A0A                          10+         SVC   10                     ISSUE GETMAIN SVC
000012 50D0 1004            00004    11          ST    13,4(,1)          Store back chain
000016 5010 D008            00008    12          ST    1,8(,13)          Store forward chain
00001A 18D1                          13          LR    13,1              Now have local save area
                R:D  00000           14          Using WA,13             Map the work area
00001C 5820 2000            00000    15          L     2,0(,2)           Get argument address
000020 5810 2000            00000    16          L     1,0(,2)           Get argument N
000024 A71E 0002            00002    17          CHI   1,2               Is the argument 2?
000028 A784 0016            00054    18          JE    Ret               Fact(2) = 2, no recursion
00002C A7D4 0021            0006E    19          JNH   Test              Go test for 0, 1, or < 0
000030 5010 D028            00028    20          ST    1,Arg             Save N we're called with
000034 0610                          21          BCTR  1,0               Create N-1
000036 5010 D02C            0002C    22          ST    1,NewArg          Store argument for call
00003A 4110 D02C            0002C    23          LA    1,NewArg          Create argument address
00003E 5010 D030            00030    24          ST    1,ArgAddr         Store the address
000042 4110 D030            00030    25          LA    1,ArgAddr         Point to argument address
000046 C0F0 FFFF FFDD       00000    26          LARL  15,RFact          Get entry point address
00004C 0DEF                          27          BASR  14,15             Call ourselves recursively
                                     28 *        Value returned in GR0
00004E 5810 D028            00028    29 Calc     L     1,Arg             Get the N we were called with
000052 1C00                          30          MR    0,0               Form N*Factorial(N-1)
000054 5820 D004            00004    31 Ret      L     2,Save+4          Get caller's save area address
                                     32          Drop  13                No more reference to work area
000058 5010 2014            00014    33          ST    1,20(,2)          Store product in R0 slot
00005C A708 0038            00038    34          LHI   0,LWA             Set work area length in GR0
                                     35          FREEMAIN R,LV=(0),A=(13) Release our storage
000060 4110 D000            00000    36+         LA    1,0(0,13)                   LOAD AREA ADDRESS
000064 0A0A                          37+         SVC   10                          ISSUE FREEMAIN SVC
                                     38          Drop  3                 No more based references here
000066 18D2                          39          LR    13,2              Restore caller's R13
000068 98E3 200C            0000C    40          LM    14,3,12(2)        Restore registers
00006C 07FE                          41          BR    14                Return to whoever called
00006E 1211                          42 Test     LTR   1,1               Is argument 0 or 1 or < 0
000070 A744 0006            0007C    43          JM    SetZ              If < 0, return 0
000074 A718 0001            00001    44          LHI   1,1               If 0 or 1, return 1
000078 A7F4 FFEE            00054    45          J     Ret               And complete return sequence
00007C 1711                          46 SetZ     XR    1,1               Return 0 for N < 0
00007E A7F4 FFEB            00054    47          J     Ret               And return
```

Figure 782 (Part 1 of 2). Assembly listing of the reenterable recursive routine

```
000000              00000 00038    49 WA       DSect ,                  Work area mapping
000000                             50 Save     DS    5D                 Local save area, GR14-GR3
000028                             51 Arg      DS    F                  Value we were called with
00002C                             52 NewArg   DS    F                  Argument for recursive call
000030                             53 ArgAddr  DS    A                  Address of the argument
000038                             54          DS    0D                 Round to doubleword
                    00038          55 LWA      Equ   *-WA               Length of work area
                                   56          End
```

Figure 782 (Part 2 of 2). Assembly listing of the reenterable recursive routine

Several items are worth noting:

1. The routine tries to use a minimum number of general registers, while maintaining standard linkage and save area conventions. (This contributes nothing to efficient performance for this particular problem; an iterative solution would be much more efficient.)
2. The local base register in GR3 is needed only because the expansion of the GETMAIN macro must generate a BAL instruction, as we saw in Figure 748 on page 959. In some situations, the FREEMAIN macro may also generate instructions needing base-displacement resolution.
3. Returning the function value in GR0 can be done by reloading the other registers; storing the result into the caller's GR0 slot in his save area is another method.

Exercises

42.2.1.(2)+ In Figure 781 on page 990, give two reasons why we can't use `BAS 14,RFACT` to cause recursion, rather than the LR and BASR instructions in the figure? Both will go to the same place.

42.2.2.(2) If you call `RFact` in Figure 781 on page 990 with argument 5, show (a) the ordered recursive calls with their arguments, and (b) the steps followed on recursion returns to evaluate the final result.

42.2.3.(3)+ By studying Figure 782 on page 991, what is the “earliest” position to which you can safely move the DROP 3 statement?

42.2.4.(2)+ Write a “main” program to call and test `RFact` example in Figure 778 on page 985.

42.3. Summary

The general principles for writing reenterable programs are fairly simple:

- All local work areas must either be in acquired storage, or in storage provided by the caller. If acquired, the storage must be released on exit from your routine.
- The program must not modify itself.
- Note that some IBM macros make it difficult to write reenterable programs, because their expansions generate in-line data areas into which the macro stores argument values. In such cases, use the List and Execute forms of the macros.

Terms and Definitions

recursion
: (1) A mathematical function defined in terms of itself. (2) A program that calls itself either directly or indirectly. Such a program may or may not be reenterable.

reenterability
: The ability of a program to be interrupted, and then executed by pne or more different programs, with no disruption to any program when it resumes execution.

reentrant
A mathematical property of certain curves. Sometimes used when *reenterability* is meant.

Programming Problems

Problem 42.1.(4)+ Write a non-reenterable recursive subroutine with a single fullword argument N that evaluates N factorial and returns its value in GR0. Then write a program to call it with values from 12 to 1, and display the results to verify its execution.

Problem 42.2.(2)+ You may have noticed that the program in Figure 780 on page 986 omitted any FREEPOOL macros. Rewrite, test, and run the program with the proper FREEPOOL macros.

Problem 42.3.(1)+ Create your own version of the factorial routine in Figure 781 on page 990, and then create a calling program to request values of N! for values of N from 1 to 12. What happens on your system if you try calculating Factorial(13)?

Problem 42.4.(2)+ Write a program to calculate the seventh Fibonacci number recursively. Include enough tracing output so you can see the great labors performed to achieve a simple result. (That is, FIBO(7) requires calculating FIBO(6) and FIBO(5); but FIBO(6) requires FIBO(5) (again) and FIBO(4), and FIBO(5) requires FIBO(4) (again) and FIBO(3), etc. Many values are calculated repeatedly.)

Problem 42.5.(2)+ The binomial coefficients can be defined both iteratively and recursively; the recursive definition is for $1 \leq N \leq M$:

```
BINCO(N,M) = 1 if M = 0 or M = N
BINCO(N,M) = BINCO(N-1,M-1) + BINCO(N-1,M) otherwise
```

Write a program to evaluate the binomial coefficients up to M=10.

Problem 42.6.(3)+ The definition above of the remainder function REMDR may seem unnecessarily complex. To show why it can be a great simplification, write a recursive program to evaluate the remainder when
(1) A=24, b=48, C=11 (that is, the remainder of $24^{48}/11$), and
(2) A=73, b=51, C=16 (that is, REMDR($73^{51}/16$)).

Problem 42.7.(4)+ The partition function PART1(N) requires evaluating the R expressions for each N until successive values are zero. Write a recursive program to evaluate PART1(N) for N from 5 to 9.

Problem 42.8.(3)+ The partition function PART2(N,M) defined on page 988 is easier to evaluate than the PART1(N) in Problem 42.7. Write a recursive Evaluate PART2(N,M) for M from 5 to 9 and for N from 1 to M. Compare your values of PART2(N,N) to the values of PART1(N) you calculated in Problem 42.7.

Problem 42.9.(3)+ The PART2(N,M) values of the partition function can be calculated iteratively using this algorithm:

```
array PART2(1:M,1:N)
for k=1 step 1 until M do
    begin for j=1 step 1 until N do
          PART2(k,j) = if k=1 or j=1 then 1 else
                       if k≤j then 1+PART2(k,j-1) else
                       PART2(k,j-1) + PART2(k-j,j)
```

Write a program to evaluate the Q array up to PART2(9,9), and compare your results to the values you calculated in Problem 42.8. (Note that no element of the PART2 array is calculated more than once.)

Problem 42.10.(3)+ Write a recursive program using the formulae for evaluating REMDR(A^b/C) to read values for A, b, and C including at least these sets of three values:

- `A =  73, b =  23, C = 109`
- `A =  73, b =  79, C = 127`
- `A = 143, b = 204, C = 999`

Problem 42.11.(3)+ Write a recursive program to evaluate Ackermann's function ACK(3,3). Your answer should be 61. (Be careful about exploring larger arguments: the function grows extremely rapidly!)

Problem 42.12.(3)+ Write a program to solve recursively the “Tower of Hanoi” problem described above. If you can display or animate the steps of the algorithm, so much the better.

Problem 42.13.(3)+ Write a program to solve the “Tower of Hanoi” problem described above on page 989, using this *iterative* method.[316]

- Number the disks from 1 to N, smallest to largest.
- Order the posts so that moving clockwise and counterclockwise are meaningful.

Then:

- Move odd-numbered disks only clockwise and even-numbered disks only counterclockwise.
- Don't move the same disk twice in succession.
- Don't put a larger disk on a smaller.

[316] Due to Wm. Randolph Franklin, published in SIGPLAN Notices Aug. 1984, page 87.

Appendix A: Conversion and Reference Tables

```
 CCCCCCCCCC   RRRRRRRRRRR   TTTTTTTTTTTT
CCCCCCCCCCCC  RRRRRRRRRRRR  TTTTTTTTTTTT
CC        CC  RR         RR      TT
CC            RR         RR      TT
CC            RR         RR      TT
CC            RRRRRRRRRRRR       TT
CC            RRRRRRRRRRR        TT
CC            RR     RR          TT
CC            RR      RR         TT
CC        CC  RR       RR        TT
CCCCCCCCCCCC  RR        RR       TT
 CCCCCCCCCC   RR         RR      TT
```

This appendix contains the following tables:

1. Table 438 gives the correspondences between hexadecimal digits and their decimal and binary representations.
2. Tables 439 and 440 are hexadecimal addition and multiplication tables.
3. Tables 441 and 442 give powers of 2 from 1 to 64 and from 65 to 128, respectively.
4. Tables 443 and 444 provide multiples of powers of 16, up to 15×16^7
5. Table 445 shows the powers of 10 expressed in hexadecimal.
6. The tables in "Hexadecimal and Decimal Integers" on page 1003 provide rapid conversions between decimal and hexadecimal for integers between 0 and 4095.
7. The tables in "Conversion Tables for Hexadecimal Fractions" on page 1011 provide conversions between decimal and hexadecimal fractions.
8. "EBCDIC Character Representation in Assembler Language Programs" on page 1012 shows the encodings used by the Assembler for EBCDIC characters.
9. "ASCII Character Representation in Assembler Language Programs" on page 1013 shows the encodings used by the Assembler for ASCII characters.
10. "DC Statement Types" on page 1014 shows the types of constants supported by the DC and DS statements.

Hexadecimal Digits in Decimal and Binary

Table 438. Hexadecimal, decimal, and binary

Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal Addition and Multiplication Tables

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Table 439. Hexadecimal Addition Table

×	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	00	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Table 440. Hexadecimal Multiplication Table

Powers of 2

Table 441. Integer powers of 2

	2^N	N
	2	1
	4	2
	8	3
	16	4
	32	5
	64	6
	128	7
	256	8
	512	9
K (kilo)	1,024	10
	2,048	11
	4,096	12
	8,192	13
	16,384	14
	32,768	15
	65,536	16
	131,072	17
	262,144	18
	524,288	19
M (mega)	1,048,576	20
	2,097,152	21
	4,194,304	22
	8,388,608	23
	16,777,216	24
	33,554,432	25
	67,108,864	26
	134,217,728	27
	268,435,456	28
	536,870,912	29
G (giga)	1,073,741,824	30
	2,147,483,648	31
	4,294,967,296	32

Table 441. Integer powers of 2

	2^N	N
	8,589,934,592	33
	17,179,869,184	34
	34,359,738,368	35
	68,719,476,736	36
	137,438,953,472	37
	274,877,906,944	38
	549,755,813,888	39
T (tera)	1,099,511,627,776	40
	2,199,023,255,552	41
	4,398,046,511,104	42
	8,796,093,022,208	43
	17,592,186,044,416	44
	35,184,372,088,832	45
	70,368,744,177,664	46
	140,737,488,355,328	47
	281,474,976,710,656	48
	562,949,953,421,312	49
P (peta)	1,125,899,906,842,624	50
	2,251,799,813,685,248	51
	4,503,599,627,370,496	52
	9,007,199,254,740,992	53
	18,014,398,509,481,984	54
	36,028,797,018,963,968	55
	72,057,594,037,927,936	56
	144,115,188,075,855,872	57
	288,230,376,151,711,744	58
	576,460,752,303,423,488	59
E (exa)	1,152,921,504,606,846,976	60
	2,305,843,009,213,693,952	61
	4,611,686,018,427,387,904	62
	9,223,372,036,854,775,808	63
	18,446,744,073,709,551,616	64

Table 442 (Page 1 of 2). Integer powers of 2		
	2^N	N
	36,893,488,147,419,103,232	65
	73,786,976,294,838,206,464	66
	147,573,952,589,676,412,928	67
	295,147,905,179,352,825,856	68
	590,295,810,358,705,651,712	69
Z (zetta)	1,180,591,620,717,411,303,424	70
	2,361,183,241,434,822,606,848	71
	4,722,366,482,869,645,213,696	72
	9,444,732,965,739,290,427,392	73
	18,889,465,931,478,580,854,784	74
	37,778,931,862,957,161,709,568	75
	75,557,863,725,914,323,419,136	76
	151,115,727,451,828,646,838,272	77
	302,231,454,903,657,293,676,544	78
	604,462,909,807,314,587,353,088	79
Y (yotta)	1,208,925,819,614,629,174,706,176	80
	2,417,851,639,229,258,349,412,352	81
	4,835,703,278,458,516,698,824,704	82
	9,671,406,556,917,033,397,649,408	83
	19,342,813,113,834,066,795,298,816	84
	38,685,626,227,668,133,590,597,632	85
	77,371,252,455,336,267,181,195,264	86
	154,742,504,910,672,534,362,390,528	87
	309,485,009,821,345,068,724,781,056	88
	618,970,019,642,690,137,449,562,112	89
	1,237,940,039,285,380,274,899,124,224	90
	2,475,880,078,570,760,549,798,248,448	91
	4,951,760,157,141,521,099,596,496,896	92
	9,903,520,314,283,042,199,192,993,792	93
	19,807,040,628,566,084,398,385,987,584	94
	39,614,081,257,132,168,796,771,975,168	95
	79,228,162,514,264,337,593,543,950,336	96

Table 442 (Page 2 of 2). Integer powers of 2	
2^N	N
158,456,325,028,528,675,187,087,900,672	97
316,912,650,057,057,350,374,175,801,344	98
633,825,300,114,114,700,748,351,602,688	99
1,267,650,600,228,229,401,496,703,205,376	100
2,535,301,200,456,458,802,993,406,410,752	101
5,070,602,400,912,917,605,986,812,821,504	102
10,141,204,801,825,835,211,973,625,643,008	103
20,282,409,603,651,670,423,947,251,286,016	104
40,564,819,207,303,340,847,894,502,572,032	105
81,129,638,414,606,681,695,789,005,144,064	106
162,259,276,829,213,363,391,578,010,288,128	107
324,518,553,658,426,726,783,156,020,576,256	108
649,037,107,316,853,453,566,312,041,152,512	109
1,298,074,214,633,706,907,132,624,082,305,024	110
2,596,148,429,267,413,814,265,248,164,610,048	111
5,192,296,858,534,827,628,530,496,329,220,096	112
10,384,593,717,069,655,257,060,992,658,440,192	113
20,769,187,434,139,310,514,121,985,316,880,384	114
41,538,374,868,278,621,028,243,970,633,760,768	115
83,076,749,736,557,242,056,487,941,267,521,536	116
166,153,499,473,114,484,112,975,882,535,043,072	117
332,306,998,946,228,968,225,951,765,070,086,144	118
664,613,997,892,457,936,451,903,530,140,172,288	119
1,329,227,995,784,915,872,903,807,060,280,344,576	120
2,658,455,991,569,831,745,807,614,120,560,689,152	121
5,316,911,983,139,663,491,615,228,241,121,378,304	122
10,633,823,966,279,326,983,230,456,482,242,756,608	123
21,267,647,932,558,653,966,460,912,964,485,513,216	124
42,535,295,865,117,307,932,921,825,928,971,026,432	125
85,070,591,730,234,615,865,843,651,857,942,052,864	126
170,141,183,460,469,231,731,687,303,715,884,105,728	127
340,282,366,920,938,463,463,374,607,431,768,211,456	128

Multiples of Powers of Sixteen

Table 443. Multiples of powers of sixteen (part 1 of 2)

Hex Digit	× 16^0	× 16^1	× 16^2	× 16^3	× 16^4
1	1	16	256	4,096	65,536
2	2	32	512	8,192	131,072
3	3	48	768	12,288	196,608
4	4	64	1,024	16,384	262,144
5	5	80	1,280	20,480	327,680
6	6	96	1,536	24,576	393,216
7	7	112	1,792	28,672	458,752
8	8	128	2,048	32,768	524,288
9	9	144	2,304	36,864	589,824
A	10	160	2,560	40,960	655,360
B	11	176	2,816	45,056	720,896
C	12	192	3,072	49,152	786,432
D	13	208	3,328	53,248	851,968
E	14	224	3,584	57,344	917,504
F	15	240	3,840	61,440	983,040

Table 444. Multiples of powers of sixteen (part 2 of 2)

Hex Digit	× 16^5	× 16^6	× 16^7
1	1,048,576	16,777,216	268,435,456
2	2,097,152	33,554,432	536,870,912
3	3,145,728	50,331,648	805,306,368
4	4,194,304	67,108,864	1,073,741,824
5	5,242,880	83,886,080	1,342,177,280
6	6,291,456	100,663,296	1,610,612,736
7	7,340,032	117,440,512	1,879,048,192
8	8,388,608	134,217,728	2,147,483,648
9	9,437,184	150,994,944	2,415,919,104
A	10,485,760	167,772,160	2,684,354,560
B	11,534,336	184,549,376	2,952,790,016
C	12,582,912	201,326,592	3,221,225,472
D	13,631,488	218,103,808	3,489,660,928
E	14,680,064	234,881,024	3,758,096,384
F	15,728,640	251,658,240	4,026,531,840

Powers of 10 in Hexadecimal

10^N in hexadecimal	N
1	0
A	1
64	2
3E8	3
2710	4
186A0	5
F4240	6
989680	7
5F5E100	8
3B9ACA00	9
2540BE400	10
174876E800	11
E8D4A51000	12
9184E72A000	13
5AF3107A4000	14
38D7EA4C68000	15
2386F26FC10000	16
16345785D8A0000	17
DE0B6B3A7640000	18
8AC7230489E80000	19
56BC75E2D63100000	20
3635C9ADC5DEA00000	21
21E19E0C9BAB2400000	22
152D02C7E14AF6800000	23
D3C21BCECCEDA1000000	24
84595161401484A000000	25
52B7D2DCC80CD2E4000000	26
33B2E3C9FD0803CE8000000	27
204FCE5E3E25026110000000	28
1431E0FAE6D7217CAA0000000	29
C9F2C9CD04674EDEA40000000	30
7E37BE2022C0914B2680000000	31
4EE2D6D415B85ACEF8100000000	32
314DC6448D9338C15B0A00000000	33
1ED09BEAD87C0378D8E6400000000	34
13426172C74D822B878FE800000000	35
C097CE7BC90715B34B9F1000000000	36
785EE10D5DA46D900F436A000000000	37
4B3B4CA85A86C47A098A224000000000	38
2F050FE938943ACC45F65568000000000	39
1D6329F1C35CA4BFABB9F5610000000000	40
125DFA371A19E6F7CB54395CA0000000000	41
B7ABC627050305ADF14A3D9E40000000000	42
72CB5BD86321E38CB6CE6682E80000000000	43
47BF19673DF52E37F2410011D100000000000	44
2CD76FE086B93CE2F768A00B22A00000000000	45
1C06A5EC5433C60DDAA16406F5A400000000000	46
118427B3B4A05BC8A8A4DE845986800000000000	47
AF298D050E4395D69670B12B7F41000000000000	48
6D79F82328EA3DA61E066EBB2F88A000000000000	49
446C3B15F9926687D2C40534FDB564000000000000	50

10^N in hexadecimal	N
2AC3A4EDBBFB8014E3BA83411E915E8000000000000	51
1ABA4714957D300D0E549208B31ADB10000000000000	52
10B46C6CDD6E3E0828F4DB456FF0C8EA0000000000000	53
A70C3C40A64E6C51999090B65F67D9240000000000000	54
6867A5A867F103B2FFFA5A71FBA0E7B680000000000000	55
4140C78940F6A24FDFFC78873D4490D2100000000000000	56
28C87CB5C89A2571EBFDCB54864ADA834A00000000000000	57
197D4DF19D605767337E9F14D3EEC8920E400000000000000	58
FEE50B7025C36A0802F236D04753D5B48E800000000000000	59
9F4F2726179A224501D762422C946590D91000000000000000	60
63917877CEC0556B21269D695BDCBF7A87AA000000000000000	61
3E3AEB4AE1383562F4B82261D969F7AC94CA4000000000000000	62
26E4D30ECCC3215DD8F3157D27E23ACBDCFE68000000000000000	63
184F03E93FF9F4DAA797ED6E38ED64BF6A1F010000000000000000	64
F316271C7FC3908A8BEF464E3945EF7A25360A0000000000000000	65
97EDD871CFDA3A5697758BF0E3CBB5AC5741C640000000000000000	66
5EF4A74721E864761EA977768E5F518BB6891BE80000000000000000	67
3B58E88C75313EC9D329EAAA18FB92F75215B17100000000000000000	68
25179157C93EC73E23FA32AA4F9D3BDA934D8EE6A00000000000000000	69
172EBAD6DDC73C86D67C5FAA71C245689C1079502400000000000000000	70
E7D34C64A9C85D4460DBBCA87196B61618A4BD216800000000000000000	71
90E40FBEEA1D3A4ABC8955E946FE31CDCF66F634E1000000000000000000	72
5A8E89D75252446EB5D5D5B1CC5EDF20A1A059E10CA000000000000000000	73
3899162693736AC531A5A58F1FBB4B746504382CA7E4000000000000000000	74
235FADD81C2822BB3F07877973D50F28BF22A31BE8EE8000000000000000000	75
161BCCA7119915B50764B4ABE86529797775A5F1719510000000000000000000	76
DD15FE86AFFAD91249EF0EB713F39EBEAA987B6E6FD2A0000000000000000000	77
8A2DBF142DFCC7AB6E3569326C7843372A9F4D2505E3A40000000000000000000	78
565C976C9CBDFCCB24E161BF83CB2A027AA3903723AE4680000000000000000000	79
35F9DEA3E1F6BDFEF70CDD17B25EFA418CA63A22764CEC100000000000000000000	80
21BC2B266D3A36BF5A680A2ECF7B5C68F7E7E45589F0138A00000000000000000000	81
15159AF8044462379881065D41AD19C19AF0EEB576360C36400000000000000000000	82
D2D80DB02AABD62BF50A3FA490C301900D6953169E1C7A1E800000000000000000000	83
83C7088E1AAB65DB792667C6DA79E0FA0861D3EE22D1CC531000000000000000000000	84
525C6558D0AB1FA92BB800DC488C2C9C453D2474D5C31FB3EA000000000000000000000	85
3379BF57826AF3C9BB530089AD579BE1AB4636C90599F3D0724000000000000000000000	86
202C1796B182D85E1513E0560C56C16D0B0BE23DA38038624768000000000000000000000	87
141B8EBE2EF1C73ACD2C6C35C7B638E426E76D668630233D6CA10000000000000000000000	88
C913936DD571C84C03BC3A19CD1E38E9850A46013DE160663E4A0000000000000000000000	89
7DAC3C24A5671D2F8255A4502032E391F3266BC0C6ACDC3FE6EE40000000000000000000000	90
4E8BA596E760723DB17586B2141FCE3B37F803587C2C09A7F054E80000000000000000000000	91
3117477E509C47668EE9742F4C93E0E502FB02174D9B8608F6351100000000000000000000000	92
1EAE8CAEF261ACA01951E89D8FDC6C8F21DCE14E908133C599E12AA00000000000000000000000	93
132D17ED577D0BE40FD3316279E9C3D9752A0CD11A50C05B802CBAA400000000000000000000000	94
BFC2EF456AE276E89E3FEDD8C321A67E93A4802B0727839301BF4A6800000000000000000000000	95
77D9D58B62CD8A5162E7F4A779F5080F1C46D01AE478B23BE1178E81000000000000000000000000	96
4AE825771DC07672DDD0F8E8AC39250971AC4210CECB6F656CAEB910A000000000000000000000000	97
2ED1176A72984A07CAA29B916BA3B725E70BA94A813F259F63ED33AA64000000000000000000000000	98
1D42AEA2879F2E44DEA5A13AE3465277B06749CE90C777839E74404A7E8000000000000000000000000	99
1249AD2594C37CEB0B2784C4CE0BF38ACE408E211A7CAAB24308A82E8F10000000000000000000000000	100

Table 445. Powers of 10 expressed in hexadecimal

Hexadecimal and Decimal Integers

These tables convert integers between 0 and 4095 (X'000' and X'FFF'). For example, to convert X'123' to decimal, find the first two digits (12) in the column headed 12*, and then find the row numbered 3. At the intersection you will find the decimal value 291.

	00*	01*	02*	03*	04*	05*	06*	07*
0	0	16	32	48	64	80	96	112
1	1	17	33	49	65	81	97	113
2	2	18	34	50	66	82	98	114
3	3	19	35	51	67	83	99	115
4	4	20	36	52	68	84	100	116
5	5	21	37	53	69	85	101	117
6	6	22	38	54	70	86	102	118
7	7	23	39	55	71	87	103	119
8	8	24	40	56	72	88	104	120
9	9	25	41	57	73	89	105	121
A	10	26	42	58	74	90	106	122
B	11	27	43	59	75	91	107	123
C	12	28	44	60	76	92	108	124
D	13	29	45	61	77	93	109	125
E	14	30	46	62	78	94	110	126
F	15	31	47	63	79	95	111	127

	08*	09*	0A*	0B*	0C*	0D*	0E*	0F*
0	128	144	160	176	192	208	224	240
1	129	145	161	177	193	209	225	241
2	130	146	162	178	194	210	226	242
3	131	147	163	179	195	211	227	243
4	132	148	164	180	196	212	228	244
5	133	149	165	181	197	213	229	245
6	134	150	166	182	198	214	230	246
7	135	151	167	183	199	215	231	247
8	136	152	168	184	200	216	232	248
9	137	153	169	185	201	217	233	249
A	138	154	170	186	202	218	234	250
B	139	155	171	187	203	219	235	251
C	140	156	172	188	204	220	236	252
D	141	157	173	189	205	221	237	253
E	142	158	174	190	206	222	238	254
F	143	159	175	191	207	223	239	255

	10*	11*	12*	13*	14*	15*	16*	17*
0	256	272	288	304	320	336	352	368
1	257	273	289	305	321	337	353	369
2	258	274	290	306	322	338	354	370
3	259	275	291	307	323	339	355	371
4	260	276	292	308	324	340	356	372
5	261	277	293	309	325	341	357	373
6	262	278	294	310	326	342	358	374
7	263	279	295	311	327	343	359	375
8	264	280	296	312	328	344	360	376
9	265	281	297	313	329	345	361	377
A	266	282	298	314	330	346	362	378
B	267	283	299	315	331	347	363	379
C	268	284	300	316	332	348	364	380
D	269	285	301	317	333	349	365	381
E	270	286	302	318	334	350	366	382
F	271	287	303	319	335	351	367	383

	18*	19*	1A*	1B*	1C*	1D*	1E*	1F*
0	384	400	416	432	448	464	480	496
1	385	401	417	433	449	465	481	497
2	386	402	418	434	450	466	482	498
3	387	403	419	435	451	467	483	499
4	388	404	420	436	452	468	484	500
5	389	405	421	437	453	469	485	501
6	390	406	422	438	454	470	486	502
7	391	407	423	439	455	471	487	503
8	392	408	424	440	456	472	488	504
9	393	409	425	441	457	473	489	505
A	394	410	426	442	458	474	490	506
B	395	411	427	443	459	475	491	507
C	396	412	428	444	460	476	492	508
D	397	413	429	445	461	477	493	509
E	398	414	430	446	462	478	494	510
F	399	415	431	447	463	479	495	511

	20*	21*	22*	23*	24*	25*	26*	27*
0	512	528	544	560	576	592	608	624
1	513	529	545	561	577	593	609	625
2	514	530	546	562	578	594	610	626
3	515	531	547	563	579	595	611	627
4	516	532	548	564	580	596	612	628
5	517	533	549	565	581	597	613	629
6	518	534	550	566	582	598	614	630
7	519	535	551	567	583	599	615	631
8	520	536	552	568	584	600	616	632
9	521	537	553	569	585	601	617	633
A	522	538	554	570	586	602	618	634
B	523	539	555	571	587	603	619	635
C	524	540	556	572	588	604	620	636
D	525	541	557	573	589	605	621	637
E	526	542	558	574	590	606	622	638
F	527	543	559	575	591	607	623	639

	28*	29*	2A*	2B*	2C*	2D*	2E*	2F*
0	640	656	672	688	704	720	736	752
1	641	657	673	689	705	721	737	753
2	642	658	674	690	706	722	738	754
3	643	659	675	691	707	723	739	755
4	644	660	676	692	708	724	740	756
5	645	661	677	693	709	725	741	757
6	646	662	678	694	710	726	742	758
7	647	663	679	695	711	727	743	759
8	648	664	680	696	712	728	744	760
9	649	665	681	697	713	729	745	761
A	650	666	682	698	714	730	746	762
B	651	667	683	699	715	731	747	763
C	652	668	684	700	716	732	748	764
D	653	669	685	701	717	733	749	765
E	654	670	686	702	718	734	750	766
F	655	671	687	703	719	735	751	767

	30*	31*	32*	33*	34*	35*	36*	37*
0	768	784	800	816	832	848	864	880
1	769	785	801	817	833	849	865	881
2	770	786	802	818	834	850	866	882
3	771	787	803	819	835	851	867	883
4	772	788	804	820	836	852	868	884
5	773	789	805	821	837	853	869	885
6	774	790	806	822	838	854	870	886
7	775	791	807	823	839	855	871	887
8	776	792	808	824	840	856	872	888
9	777	793	809	825	841	857	873	889
A	778	794	810	826	842	858	874	890
B	779	795	811	827	843	859	875	891
C	780	796	812	828	844	860	876	892
D	781	797	813	829	845	861	877	893
E	782	798	814	830	846	862	878	894
F	783	799	815	831	847	863	879	895

	38*	39*	3A*	3B*	3C*	3D*	3E*	3F*
0	896	912	928	944	960	976	992	1008
1	897	913	929	945	961	977	993	1009
2	898	914	930	946	962	978	994	1010
3	899	915	931	947	963	979	995	1011
4	900	916	932	948	964	980	996	1012
5	901	917	933	949	965	981	997	1013
6	902	918	934	950	966	982	998	1014
7	903	919	935	951	967	983	999	1015
8	904	920	936	952	968	984	1000	1016
9	905	921	937	953	969	985	1001	1017
A	906	922	938	954	970	986	1002	1018
B	907	923	939	955	971	987	1003	1019
C	908	924	940	956	972	988	1004	1020
D	909	925	941	957	973	989	1005	1021
E	910	926	942	958	974	990	1006	1022
F	911	927	943	959	975	991	1007	1023

	40*	41*	42*	43*	44*	45*	46*	47*
0	1024	1040	1056	1072	1088	1104	1120	1136
1	1025	1041	1057	1073	1089	1105	1121	1137
2	1026	1042	1058	1074	1090	1106	1122	1138
3	1027	1043	1059	1075	1091	1107	1123	1139
4	1028	1044	1060	1076	1092	1108	1124	1140
5	1029	1045	1061	1077	1093	1109	1125	1141
6	1030	1046	1062	1078	1094	1110	1126	1142
7	1031	1047	1063	1079	1095	1111	1127	1143
8	1032	1048	1064	1080	1096	1112	1128	1144
9	1033	1049	1065	1081	1097	1113	1129	1145
A	1034	1050	1066	1082	1098	1114	1130	1146
B	1035	1051	1067	1083	1099	1115	1131	1147
C	1036	1052	1068	1084	1100	1116	1132	1148
D	1037	1053	1069	1085	1101	1117	1133	1149
E	1038	1054	1070	1086	1102	1118	1134	1150
F	1039	1055	1071	1087	1103	1119	1135	1151

	48*	49*	4A*	4B*	4C*	4D*	4E*	4F*
0	1152	1168	1184	1200	1216	1232	1248	1264
1	1153	1169	1185	1201	1217	1233	1249	1265
2	1154	1170	1186	1202	1218	1234	1250	1266
3	1155	1171	1187	1203	1219	1235	1251	1267
4	1156	1172	1188	1204	1220	1236	1252	1268
5	1157	1173	1189	1205	1221	1237	1253	1269
6	1158	1174	1190	1206	1222	1238	1254	1270
7	1159	1175	1191	1207	1223	1239	1255	1271
8	1160	1176	1192	1208	1224	1240	1256	1272
9	1161	1177	1193	1209	1225	1241	1257	1273
A	1162	1178	1194	1210	1226	1242	1258	1274
B	1163	1179	1195	1211	1227	1243	1259	1275
C	1164	1180	1196	1212	1228	1244	1260	1276
D	1165	1181	1197	1213	1229	1245	1261	1277
E	1166	1182	1198	1214	1230	1246	1262	1278
F	1167	1183	1199	1215	1231	1247	1263	1279

	50*	51*	52*	53*	54*	55*	56*	57*
0	1280	1296	1312	1328	1344	1360	1376	1392
1	1281	1297	1313	1329	1345	1361	1377	1393
2	1282	1298	1314	1330	1346	1362	1378	1394
3	1283	1299	1315	1331	1347	1363	1379	1395
4	1284	1300	1316	1332	1348	1364	1380	1396
5	1285	1301	1317	1333	1349	1365	1381	1397
6	1286	1302	1318	1334	1350	1366	1382	1398
7	1287	1303	1319	1335	1351	1367	1383	1399
8	1288	1304	1320	1336	1352	1368	1384	1400
9	1289	1305	1321	1337	1353	1369	1385	1401
A	1290	1306	1322	1338	1354	1370	1386	1402
B	1291	1307	1323	1339	1355	1371	1387	1403
C	1292	1308	1324	1340	1356	1372	1388	1404
D	1293	1309	1325	1341	1357	1373	1389	1405
E	1294	1310	1326	1342	1358	1374	1390	1406
F	1295	1311	1327	1343	1359	1375	1391	1407

	58*	59*	5A*	5B*	5C*	5D*	5E*	5F*
0	1408	1424	1440	1456	1472	1488	1504	1520
1	1409	1425	1441	1457	1473	1489	1505	1521
2	1410	1426	1442	1458	1474	1490	1506	1522
3	1411	1427	1443	1459	1475	1491	1507	1523
4	1412	1428	1444	1460	1476	1492	1508	1524
5	1413	1429	1445	1461	1477	1493	1509	1525
6	1414	1430	1446	1462	1478	1494	1510	1526
7	1415	1431	1447	1463	1479	1495	1511	1527
8	1416	1432	1448	1464	1480	1496	1512	1528
9	1417	1433	1449	1465	1481	1497	1513	1529
A	1418	1434	1450	1466	1482	1498	1514	1530
B	1419	1435	1451	1467	1483	1499	1515	1531
C	1420	1436	1452	1468	1484	1500	1516	1532
D	1421	1437	1453	1469	1485	1501	1517	1533
E	1422	1438	1454	1470	1486	1502	1518	1534
F	1423	1439	1455	1471	1487	1503	1519	1535

	60*	61*	62*	63*	64*	65*	66*	67*
0	1536	1552	1568	1584	1600	1616	1632	1648
1	1537	1553	1569	1585	1601	1617	1633	1649
2	1538	1554	1570	1586	1602	1618	1634	1650
3	1539	1555	1571	1587	1603	1619	1635	1651
4	1540	1556	1572	1588	1604	1620	1636	1652
5	1541	1557	1573	1589	1605	1621	1637	1653
6	1542	1558	1574	1590	1606	1622	1638	1654
7	1543	1559	1575	1591	1607	1623	1639	1655
8	1544	1560	1576	1592	1608	1624	1640	1656
9	1545	1561	1577	1593	1609	1625	1641	1657
A	1546	1562	1578	1594	1610	1626	1642	1658
B	1547	1563	1579	1595	1611	1627	1643	1659
C	1548	1564	1580	1596	1612	1628	1644	1660
D	1549	1565	1581	1597	1613	1629	1645	1661
E	1550	1566	1582	1598	1614	1630	1646	1662
F	1551	1567	1583	1599	1615	1631	1647	1663

	68*	69*	6A*	6B*	6C*	6D*	6E*	6F*
0	1664	1680	1696	1712	1728	1744	1760	1776
1	1665	1681	1697	1713	1729	1745	1761	1777
2	1666	1682	1698	1714	1730	1746	1762	1778
3	1667	1683	1699	1715	1731	1747	1763	1779
4	1668	1684	1700	1716	1732	1748	1764	1780
5	1669	1685	1701	1717	1733	1749	1765	1781
6	1670	1686	1702	1718	1734	1750	1766	1782
7	1671	1687	1703	1719	1735	1751	1767	1783
8	1672	1688	1704	1720	1736	1752	1768	1784
9	1673	1689	1705	1721	1737	1753	1769	1785
A	1674	1690	1706	1722	1738	1754	1770	1786
B	1675	1691	1707	1723	1739	1755	1771	1787
C	1676	1692	1708	1724	1740	1756	1772	1788
D	1677	1693	1709	1725	1741	1757	1773	1789
E	1678	1694	1710	1726	1742	1758	1774	1790
F	1679	1695	1711	1727	1743	1759	1775	1791

	70*	71*	72*	73*	74*	75*	76*	77*
0	1792	1808	1824	1840	1856	1872	1888	1904
1	1793	1809	1825	1841	1857	1873	1889	1905
2	1794	1810	1826	1842	1858	1874	1890	1906
3	1795	1811	1827	1843	1859	1875	1891	1907
4	1796	1812	1828	1844	1860	1876	1892	1908
5	1797	1813	1829	1845	1861	1877	1893	1909
6	1798	1814	1830	1846	1862	1878	1894	1910
7	1799	1815	1831	1847	1863	1879	1895	1911
8	1800	1816	1832	1848	1864	1880	1896	1912
9	1801	1817	1833	1849	1865	1881	1897	1913
A	1802	1818	1834	1850	1866	1882	1898	1914
B	1803	1819	1835	1851	1867	1883	1899	1915
C	1804	1820	1836	1852	1868	1884	1900	1916
D	1805	1821	1837	1853	1869	1885	1901	1917
E	1806	1822	1838	1854	1870	1886	1902	1918
F	1807	1823	1839	1855	1871	1887	1903	1919

	78*	79*	7A*	7B*	7C*	7D*	7E*	7F*
0	1920	1936	1952	1968	1984	2000	2016	2032
1	1921	1937	1953	1969	1985	2001	2017	2033
2	1922	1938	1954	1970	1986	2002	2018	2034
3	1923	1939	1955	1971	1987	2003	2019	2035
4	1924	1940	1956	1972	1988	2004	2020	2036
5	1925	1941	1957	1973	1989	2005	2021	2037
6	1926	1942	1958	1974	1990	2006	2022	2038
7	1927	1943	1959	1975	1991	2007	2023	2039
8	1928	1944	1960	1976	1992	2008	2024	2040
9	1929	1945	1961	1977	1993	2009	2025	2041
A	1930	1946	1962	1978	1994	2010	2026	2042
B	1931	1947	1963	1979	1995	2011	2027	2043
C	1932	1948	1964	1980	1996	2012	2028	2044
D	1933	1949	1965	1981	1997	2013	2029	2045
E	1934	1950	1966	1982	1998	2014	2030	2046
F	1935	1951	1967	1983	1999	2015	2031	2047

	80*	81*	82*	83*	84*	85*	86*	87*
0	2048	2064	2080	2096	2112	2128	2144	2160
1	2049	2065	2081	2097	2113	2129	2145	2161
2	2050	2066	2082	2098	2114	2130	2146	2162
3	2051	2067	2083	2099	2115	2131	2147	2163
4	2052	2068	2084	2100	2116	2132	2148	2164
5	2053	2069	2085	2101	2117	2133	2149	2165
6	2054	2070	2086	2102	2118	2134	2150	2166
7	2055	2071	2087	2103	2119	2135	2151	2167
8	2056	2072	2088	2104	2120	2136	2152	2168
9	2057	2073	2089	2105	2121	2137	2153	2169
A	2058	2074	2090	2106	2122	2138	2154	2170
B	2059	2075	2091	2107	2123	2139	2155	2171
C	2060	2076	2092	2108	2124	2140	2156	2172
D	2061	2077	2093	2109	2125	2141	2157	2173
E	2062	2078	2094	2110	2126	2142	2158	2174
F	2063	2079	2095	2111	2127	2143	2159	2175

	88*	89*	8A*	8B*	8C*	8D*	8E*	8F*
0	2176	2192	2208	2224	2240	2256	2272	2288
1	2177	2193	2209	2225	2241	2257	2273	2289
2	2178	2194	2210	2226	2242	2258	2274	2290
3	2179	2195	2211	2227	2243	2259	2275	2291
4	2180	2196	2212	2228	2244	2260	2276	2292
5	2181	2197	2213	2229	2245	2261	2277	2293
6	2182	2198	2214	2230	2246	2262	2278	2294
7	2183	2199	2215	2231	2247	2263	2279	2295
8	2184	2200	2216	2232	2248	2264	2280	2296
9	2185	2201	2217	2233	2249	2265	2281	2297
A	2186	2202	2218	2234	2250	2266	2282	2298
B	2187	2203	2219	2235	2251	2267	2283	2299
C	2188	2204	2220	2236	2252	2268	2284	2300
D	2189	2205	2221	2237	2253	2269	2285	2301
E	2190	2206	2222	2238	2254	2270	2286	2302
F	2191	2207	2223	2239	2255	2271	2287	2303

	90*	91*	92*	93*	94*	95*	96*	97*
0	2304	2320	2336	2352	2368	2384	2400	2416
1	2305	2321	2337	2353	2369	2385	2401	2417
2	2306	2322	2338	2354	2370	2386	2402	2418
3	2307	2323	2339	2355	2371	2387	2403	2419
4	2308	2324	2340	2356	2372	2388	2404	2420
5	2309	2325	2341	2357	2373	2389	2405	2421
6	2310	2326	2342	2358	2374	2390	2406	2422
7	2311	2327	2343	2359	2375	2391	2407	2423
8	2312	2328	2344	2360	2376	2392	2408	2424
9	2313	2329	2345	2361	2377	2393	2409	2425
A	2314	2330	2346	2362	2378	2394	2410	2426
B	2315	2331	2347	2363	2379	2395	2411	2427
C	2316	2332	2348	2364	2380	2396	2412	2428
D	2317	2333	2349	2365	2381	2397	2413	2429
E	2318	2334	2350	2366	2382	2398	2414	2430
F	2319	2335	2351	2367	2383	2399	2415	2431

	98*	99*	9A*	9B*	9C*	9D*	9E*	9F*
0	2432	2448	2464	2480	2496	2512	2528	2544
1	2433	2449	2465	2481	2497	2513	2529	2545
2	2434	2450	2466	2482	2498	2514	2530	2546
3	2435	2451	2467	2483	2499	2515	2531	2547
4	2436	2452	2468	2484	2500	2516	2532	2548
5	2437	2453	2469	2485	2501	2517	2533	2549
6	2438	2454	2470	2486	2502	2518	2534	2550
7	2439	2455	2471	2487	2503	2519	2535	2551
8	2440	2456	2472	2488	2504	2520	2536	2552
9	2441	2457	2473	2489	2505	2521	2537	2553
A	2442	2458	2474	2490	2506	2522	2538	2554
B	2443	2459	2475	2491	2507	2523	2539	2555
C	2444	2460	2476	2492	2508	2524	2540	2556
D	2445	2461	2477	2493	2509	2525	2541	2557
E	2446	2462	2478	2494	2510	2526	2542	2558
F	2447	2463	2479	2495	2511	2527	2543	2559

	A0*	A1*	A2*	A3*	A4*	A5*	A6*	A7*
0	2560	2576	2592	2608	2624	2640	2656	2672
1	2561	2577	2593	2609	2625	2641	2657	2673
2	2562	2578	2594	2610	2626	2642	2658	2674
3	2563	2579	2595	2611	2627	2643	2659	2675
4	2564	2580	2596	2612	2628	2644	2660	2676
5	2565	2581	2597	2613	2629	2645	2661	2677
6	2566	2582	2598	2614	2630	2646	2662	2678
7	2567	2583	2599	2615	2631	2647	2663	2679
8	2568	2584	2600	2616	2632	2648	2664	2680
9	2569	2585	2601	2617	2633	2649	2665	2681
A	2570	2586	2602	2618	2634	2650	2666	2682
B	2571	2587	2603	2619	2635	2651	2667	2683
C	2572	2588	2604	2620	2636	2652	2668	2684
D	2573	2589	2605	2621	2637	2653	2669	2685
E	2574	2590	2606	2622	2638	2654	2670	2686
F	2575	2591	2607	2623	2639	2655	2671	2687

	A8*	A9*	AA*	AB*	AC*	AD*	AE*	AF*
0	2688	2704	2720	2736	2752	2768	2784	2800
1	2689	2705	2721	2737	2753	2769	2785	2801
2	2690	2706	2722	2738	2754	2770	2786	2802
3	2691	2707	2723	2739	2755	2771	2787	2803
4	2692	2708	2724	2740	2756	2772	2788	2804
5	2693	2709	2725	2741	2757	2773	2789	2805
6	2694	2710	2726	2742	2758	2774	2790	2806
7	2695	2711	2727	2743	2759	2775	2791	2807
8	2696	2712	2728	2744	2760	2776	2792	2808
9	2697	2713	2729	2745	2761	2777	2793	2809
A	2698	2714	2730	2746	2762	2778	2794	2810
B	2699	2715	2731	2747	2763	2779	2795	2811
C	2700	2716	2732	2748	2764	2780	2796	2812
D	2701	2717	2733	2749	2765	2781	2797	2813
E	2702	2718	2734	2750	2766	2782	2798	2814
F	2703	2719	2735	2751	2767	2783	2799	2815

	B0*	B1*	B2*	B3*	B4*	B5*	B6*	B7*
0	2816	2832	2848	2864	2880	2896	2912	2928
1	2817	2833	2849	2865	2881	2897	2913	2929
2	2818	2834	2850	2866	2882	2898	2914	2930
3	2819	2835	2851	2867	2883	2899	2915	2931
4	2820	2836	2852	2868	2884	2900	2916	2932
5	2821	2837	2853	2869	2885	2901	2917	2933
6	2822	2838	2854	2870	2886	2902	2918	2934
7	2823	2839	2855	2871	2887	2903	2919	2935
8	2824	2840	2856	2872	2888	2904	2920	2936
9	2825	2841	2857	2873	2889	2905	2921	2937
A	2826	2842	2858	2874	2890	2906	2922	2938
B	2827	2843	2859	2875	2891	2907	2923	2939
C	2828	2844	2860	2876	2892	2908	2924	2940
D	2829	2845	2861	2877	2893	2909	2925	2941
E	2830	2846	2862	2878	2894	2910	2926	2942
F	2831	2847	2863	2879	2895	2911	2927	2943

	B8*	B9*	BA*	BB*	BC*	BD*	BE*	BF*
0	2944	2960	2976	2992	3008	3024	3040	3056
1	2945	2961	2977	2993	3009	3025	3041	3057
2	2946	2962	2978	2994	3010	3026	3042	3058
3	2947	2963	2979	2995	3011	3027	3043	3059
4	2948	2964	2980	2996	3012	3028	3044	3060
5	2949	2965	2981	2997	3013	3029	3045	3061
6	2950	2966	2982	2998	3014	3030	3046	3062
7	2951	2967	2983	2999	3015	3031	3047	3063
8	2952	2968	2984	3000	3016	3032	3048	3064
9	2953	2969	2985	3001	3017	3033	3049	3065
A	2954	2970	2986	3002	3018	3034	3050	3066
B	2955	2971	2987	3003	3019	3035	3051	3067
C	2956	2972	2988	3004	3020	3036	3052	3068
D	2957	2973	2989	3005	3021	3037	3053	3069
E	2958	2974	2990	3006	3022	3038	3054	3070
F	2959	2975	2991	3007	3023	3039	3055	3071

	C0*	C1*	C2*	C3*	C4*	C5*	C6*	C7*
0	3072	3088	3104	3120	3136	3152	3168	3184
1	3073	3089	3105	3121	3137	3153	3169	3185
2	3074	3090	3106	3122	3138	3154	3170	3186
3	3075	3091	3107	3123	3139	3155	3171	3187
4	3076	3092	3108	3124	3140	3156	3172	3188
5	3077	3093	3109	3125	3141	3157	3173	3189
6	3078	3094	3110	3126	3142	3158	3174	3190
7	3079	3095	3111	3127	3143	3159	3175	3191
8	3080	3096	3112	3128	3144	3160	3176	3192
9	3081	3097	3113	3129	3145	3161	3177	3193
A	3082	3098	3114	3130	3146	3162	3178	3194
B	3083	3099	3115	3131	3147	3163	3179	3195
C	3084	3100	3116	3132	3148	3164	3180	3196
D	3085	3101	3117	3133	3149	3165	3181	3197
E	3086	3102	3118	3134	3150	3166	3182	3198
F	3087	3103	3119	3135	3151	3167	3183	3199

	C8*	C9*	CA*	CB*	CC*	CD*	CE*	CF*
0	3200	3216	3232	3248	3264	3280	3296	3312
1	3201	3217	3233	3249	3265	3281	3297	3313
2	3202	3218	3234	3250	3266	3282	3298	3314
3	3203	3219	3235	3251	3267	3283	3299	3315
4	3204	3220	3236	3252	3268	3284	3300	3316
5	3205	3221	3237	3253	3269	3285	3301	3317
6	3206	3222	3238	3254	3270	3286	3302	3318
7	3207	3223	3239	3255	3271	3287	3303	3319
8	3208	3224	3240	3256	3272	3288	3304	3320
9	3209	3225	3241	3257	3273	3289	3305	3321
A	3210	3226	3242	3258	3274	3290	3306	3322
B	3211	3227	3243	3259	3275	3291	3307	3323
C	3212	3228	3244	3260	3276	3292	3308	3324
D	3213	3229	3245	3261	3277	3293	3309	3325
E	3214	3230	3246	3262	3278	3294	3310	3326
F	3215	3231	3247	3263	3279	3295	3311	3327

	D0*	D1*	D2*	D3*	D4*	D5*	D6*	D7*
0	3328	3344	3360	3376	3392	3408	3424	3440
1	3329	3345	3361	3377	3393	3409	3425	3441
2	3330	3346	3362	3378	3394	3410	3426	3442
3	3331	3347	3363	3379	3395	3411	3427	3443
4	3332	3348	3364	3380	3396	3412	3428	3444
5	3333	3349	3365	3381	3397	3413	3429	3445
6	3334	3350	3366	3382	3398	3414	3430	3446
7	3335	3351	3367	3383	3399	3415	3431	3447
8	3336	3352	3368	3384	3400	3416	3432	3448
9	3337	3353	3369	3385	3401	3417	3433	3449
A	3338	3354	3370	3386	3402	3418	3434	3450
B	3339	3355	3371	3387	3403	3419	3435	3451
C	3340	3356	3372	3388	3404	3420	3436	3452
D	3341	3357	3373	3389	3405	3421	3437	3453
E	3342	3358	3374	3390	3406	3422	3438	3454
F	3343	3359	3375	3391	3407	3423	3439	3455

	D8*	D9*	DA*	DB*	DC*	DD*	DE*	DF*
0	3456	3472	3488	3504	3520	3536	3552	3568
1	3457	3473	3489	3505	3521	3537	3553	3569
2	3458	3474	3490	3506	3522	3538	3554	3570
3	3459	3475	3491	3507	3523	3539	3555	3571
4	3460	3476	3492	3508	3524	3540	3556	3572
5	3461	3477	3493	3509	3525	3541	3557	3573
6	3462	3478	3494	3510	3526	3542	3558	3574
7	3463	3479	3495	3511	3527	3543	3559	3575
8	3464	3480	3496	3512	3528	3544	3560	3576
9	3465	3481	3497	3513	3529	3545	3561	3577
A	3466	3482	3498	3514	3530	3546	3562	3578
B	3467	3483	3499	3515	3531	3547	3563	3579
C	3468	3484	3500	3516	3532	3548	3564	3580
D	3469	3485	3501	3517	3533	3549	3565	3581
E	3470	3486	3502	3518	3534	3550	3566	3582
F	3471	3487	3503	3519	3535	3551	3567	3583

	E0*	E1*	E2*	E3*	E4*	E5*	E6*	E7*
0	3584	3600	3616	3632	3648	3664	3680	3696
1	3585	3601	3617	3633	3649	3665	3681	3697
2	3586	3602	3618	3634	3650	3666	3682	3698
3	3587	3603	3619	3635	3651	3667	3683	3699
4	3588	3604	3620	3636	3652	3668	3684	3700
5	3589	3605	3621	3637	3653	3669	3685	3701
6	3590	3606	3622	3638	3654	3670	3686	3702
7	3591	3607	3623	3639	3655	3671	3687	3703
8	3592	3608	3624	3640	3656	3672	3688	3704
9	3593	3609	3625	3641	3657	3673	3689	3705
A	3594	3610	3626	3642	3658	3674	3690	3706
B	3595	3611	3627	3643	3659	3675	3691	3707
C	3596	3612	3628	3644	3660	3676	3692	3708
D	3597	3613	3629	3645	3661	3677	3693	3709
E	3598	3614	3630	3646	3662	3678	3694	3710
F	3599	3615	3631	3647	3663	3679	3695	3711

	E8*	E9*	EA*	EB*	EC*	ED*	EE*	EF*
0	3712	3728	3744	3760	3776	3792	3808	3824
1	3713	3729	3745	3761	3777	3793	3809	3825
2	3714	3730	3746	3762	3778	3794	3810	3826
3	3715	3731	3747	3763	3779	3795	3811	3827
4	3716	3732	3748	3764	3780	3796	3812	3828
5	3717	3733	3749	3765	3781	3797	3813	3829
6	3718	3734	3750	3766	3782	3798	3814	3830
7	3719	3735	3751	3767	3783	3799	3815	3831
8	3720	3736	3752	3768	3784	3800	3816	3832
9	3721	3737	3753	3769	3785	3801	3817	3833
A	3722	3738	3754	3770	3786	3802	3818	3834
B	3723	3739	3755	3771	3787	3803	3819	3835
C	3724	3740	3756	3772	3788	3804	3820	3836
D	3725	3741	3757	3773	3789	3805	3821	3837
E	3726	3742	3758	3774	3790	3806	3822	3838
F	3727	3743	3759	3775	3791	3807	3823	3839

	F0*	F1*	F2*	F3*	F4*	F5*	F6*	F7*
0	3840	3856	3872	3888	3904	3920	3936	3952
1	3841	3857	3873	3889	3905	3921	3937	3953
2	3842	3858	3874	3890	3906	3922	3938	3954
3	3843	3859	3875	3891	3907	3923	3939	3955
4	3844	3860	3876	3892	3908	3924	3940	3956
5	3845	3861	3877	3893	3909	3925	3941	3957
6	3846	3862	3878	3894	3910	3926	3942	3958
7	3847	3863	3879	3895	3911	3927	3943	3959
8	3848	3864	3880	3896	3912	3928	3944	3960
9	3849	3865	3881	3897	3913	3929	3945	3961
A	3850	3866	3882	3898	3914	3930	3946	3962
B	3851	3867	3883	3899	3915	3931	3947	3963
C	3852	3868	3884	3900	3916	3932	3948	3964
D	3853	3869	3885	3901	3917	3933	3949	3965
E	3854	3870	3886	3902	3918	3934	3950	3966
F	3855	3871	3887	3903	3919	3935	3951	3967

	F8*	F9*	FA*	FB*	FC*	FD*	FE*	FF*
0	3968	3984	4000	4016	4032	4048	4064	4080
1	3969	3985	4001	4017	4033	4049	4065	4081
2	3970	3986	4002	4018	4034	4050	4066	4082
3	3971	3987	4003	4019	4035	4051	4067	4083
4	3972	3988	4004	4020	4036	4052	4068	4084
5	3973	3989	4005	4021	4037	4053	4069	4085
6	3974	3990	4006	4022	4038	4054	4070	4086
7	3975	3991	4007	4023	4039	4055	4071	4087
8	3976	3992	4008	4024	4040	4056	4072	4088
9	3977	3993	4009	4025	4041	4057	4073	4089
A	3978	3994	4010	4026	4042	4058	4074	4090
B	3979	3995	4011	4027	4043	4059	4075	4091
C	3980	3996	4012	4028	4044	4060	4076	4092
D	3981	3997	4013	4029	4045	4061	4077	4093
E	3982	3998	4014	4030	4046	4062	4078	4094
F	3983	3999	4015	4031	4047	4063	4079	4095

Conversion Tables for Hexadecimal Fractions

The six pairs of columns give respectively a hexadecimal fraction and its decimal equivalent. For example, the decimal value of the hexadecimal fraction `.FF` is .9375+.05859375, or 0.99609375.

Hex	Decimal	Hex	Decimal	Hex	Decimal
.1	.0625	.01	.00390625	.001	.000244140625
.2	.1250	.02	.00781250	.002	.000488281250
.3	.1875	.03	.01171875	.003	.000732421875
.4	.2500	.04	.01562500	.004	.000976562500
.5	.3125	.05	.01953125	.005	.001220703125
.6	.3750	.06	.02343750	.006	.001464843750
.7	.4375	.07	.02734375	.007	.001708984375
.8	.5000	.08	.03125000	.008	.001953125000
.9	.5625	.09	.03515625	.009	.002197265625
.A	.6250	.0A	.03906250	.00A	.002441406250
.B	.6875	.0B	.04296875	.00B	.002685546875
.C	.7500	.0C	.04687500	.00C	.002929687500
.D	.8125	.0D	.05078125	.00D	.003173828125
.E	.8750	.0E	.05468750	.00E	.003417968750
.F	.9375	.0F	.05859375	.00F	.003662109375

Hex	Decimal	Hex	Decimal
.0001	.0000152587890625	.00001	.00000095367431640625
.0002	.0000305175781250	.00002	.00000190734863281250
.0003	.0000457763671875	.00003	.00000286102294921875
.0004	.0000610351562500	.00004	.00000381469726562500
.0005	.0000762939453125	.00005	.00000476837158203125
.0006	.0000915527343750	.00006	.00000572204589843750
.0007	.0001068115234375	.00007	.00000667572021484375
.0008	.0001220703125000	.00008	.00000762939453125000
.0009	.0001373291015625	.00009	.00000858306884765625
.000A	.0001525878906250	.0000A	.00000953674316406250
.000B	.0001678466796875	.0000B	.00001049041748046875
.000C	.0001831054687500	.0000C	.00001144409179687500
.000D	.0001983642578125	.0000D	.00001239776611328125
.000E	.0002136230468750	.0000E	.00001335144042968750
.000F	.0002288818359375	.0000F	.00001430511474609375

Hex	Decimal
.000001	.000000059604644775390625
.000002	.000000119209289550781250
.000003	.000000178813934326171875
.000004	.000000238418579101562500
.000005	.000000298023223876953125
.000006	.000000357627868652343750
.000007	.000000417232513427734375
.000008	.000000476837158203125000
.000009	.000000536441802978515625
.00000A	.000000596046447753906250
.00000B	.000000655651092529296875
.00000C	.000000715255737304687500
.00000D	.000000774860382080078125
.00000E	.000000834465026855468750
.00000F	.000000894069671630859375

EBCDIC Character Representation in Assembler Language Programs

This table uses IBM code page 037.

Table 446. Assembler Language EBCDIC character representation

Char	Hex	Char	Hex	Char	Hex	Char	Hex
Blank	40	e	85	y	A8	S	E2
.	4B	f	86	z	A9	T	E3
(	4D	g	87	A	C1	U	E4
+	4E	h	88	B	C2	V	E5
&	50	i	89	C	C3	W	E6
$	5B	j	91	D	C4	X	E7
*	5C	k	92	E	C5	Y	E8
)	5D	l	93	F	C6	Z	E9
-	60	m	94	G	C7	0	F0
/	61	n	95	H	C8	1	F1
,	6B	o	96	I	C9	2	F2
_	6D	p	97	J	D1	3	F3
#	7B	q	98	K	D2	4	F4
@	7C	r	99	L	D3	5	F5
'	7D	s	A2	M	D4	6	F6
=	7E	t	A3	N	D5	7	F7
a	81	u	A4	O	D6	8	F8
b	82	v	A5	P	D7	9	F9
c	83	w	A6	Q	D8		
d	84	x	A7	R	D9		

ASCII Character Representation in Assembler Language Programs

Char	Code	Char	Code	Char	Code	Char	Code
blank	20	8	38	P	50	h	68
!	21	9	39	Q	51	i	69
″	22	:	3A	R	52	j	6A
#	23	;	3B	S	53	k	6B
$	24	<	3C	T	54	l	6C
%	25	=	3D	U	55	m	6D
&	26	>	3E	V	56	n	6E
'	27	?	3F	W	57	o	6F
(	28	@	40	X	58	p	70
)	29	A	41	Y	59	q	71
*	2A	B	42	Z	5A	r	72
+	2B	C	43	[	5B	s	73
,	2C	D	44		5C	t	74
-	2D	E	45	]	5D	u	75
.	2E	F	46	^	5E	v	76
/	2F	G	47	_	5F	w	77
0	30	H	48	`	60	x	78
1	31	I	49	a	61	y	79
2	32	J	4A	b	62	z	7A
3	33	K	4B	c	63	{	7B
4	34	L	4C	d	64	|	7C
5	35	M	4D	4	65	}	7D
6	36	N	4E	f	66	˜	7E
7	37	O	4F	g	67	(none)	7F

Table 447. 7-bit ASCII character representation

DC Statement Types

Table 448. High Level Assembler DC-Statement Constant Types

Type	Subtype(s)	Default Length(s)	Modifiers	Pad/ Truncate	Constant
A	(D)	4 (8)	L	Left	Absolute or relocatable expression
B		Minimum	L	Left	Binary
C		Minimum	L	Right	EBCDIC Characters
C	A	Minimum	L	Right	ASCII characters generated
C	E	Minimum	L	Right	EBCDIC characters generated
C	U	Minimum	L	Right	Unicode characters generated
D		8	L,S,E	Right	Hexadecimal floating-point
D	B	8	L,E	Right	Binary floating-point
D	D	8	E	None	Decimal floating-point
D	H	8	L,S,E	Right	Hexadecimal floating-point (improved rounding)
E		4	L,S,E	Right	Hexadecimal floating-point
E	B	4	L,E	Right	Binary floating-point
E	D	4	E	None	Decimal floating-point
E	H	4	L,S,E	Right	Hexadecimal floating-point (improved rounding)
F	(D)	4 (8)	L,S,E	Left	Fixed-point binary
G		Minimum	L	Right	Graphic (usually Kanji)
H	(D)	2 (8)	L,S,E	Left	Fixed-point binary
J	(D)	4 (8)	L	Left	Class length
L		16	L,S,E	Right	Hexadecimal floating-point
L	B	16	L,E	Right	Binary floating-point
L	D	16	E	None	Decimal floating-point
L	H	16	L,S,E	Right	Hexadecimal floating-point (improved rounding)
P		Minimum	L	Left	Packed decimal
Q	(D)	4 (8)	L	None	DXD or Part offset
Q	Y	3	L3	None	Long-displacement DXD or Part offset
R	(D)	4 (8)	L	Left	PSECT address
S		2	L2	None	12-bit-displacement base-displacement address
S	Y	3	L3	None	Long-displacement base-displacement address
V	(D)	4 (8)	L	None	External symbol
Y		2	L	Left	Absolute or relocatable expression
Z		Minimum	L	Left	Zoned decimal

Appendix B: Simple I/O Macros

```
IIIIIIIIII            // 000000000000
IIIIIIIIII           //  000000000000
    II              //   00        00
    II             //    00        00
    II            //     00        00
    II           //      00        00
    II          //       00        00
    II         //        00        00
    II        //         00        00
    II       //          00        00
    II      //           00        00
IIIIIIIIII //            000000000000
IIIIIIIIII//             000000000000
```

The forms that I/O takes within and across z/Architecture operating systems are varied enough that it takes many books to describe them. So that you won't need to understand the I/O rules associated with a particular Operating System, the simple needs of small programs like the Programming Problems here can be satisfied by the following facilities:

1. An instruction (CONVERTI) to convert decimal characters in memory to a signed binary value in a General Purpose register.
2. An instruction (CONVERTO) to convert the contents of a General Purpose register to a string of decimal characters in memory.
3. An instruction (DUMPOUT) to generate a formatted hexadecimal dump of specified areas of memory.
4. An instruction (PRINTLIN) to print line images on a printer file, with carriage control characters and optional specification of the length of the character string to be printed.
5. An instruction (PRINTOUT) to:
   - print the value of the contents of an area of memory, giving its name as well as the value of the contents in an easily-readable format.
   - print the contents of the General Purpose and Floating-Point registers.
   - return control to the Supervisor when program execution has been successfully completed.
6. An instruction (READCARD) to read 80-character records into a named area in your program, with provision for optionally transferring control to some out-of-line location if no further records are available.

These macro instructions do not change the Condition Code.

B.1. Macro Facilities

The arguments of the CONVERTI, CONVERTO, DUMPOUT, PRINTLIN, PRINTOUT, and READCARD macro-instructions use these notational abbreviations in their descriptions.

<name> any valid symbol naming an area of memory which is addressable from the point where it is used in a macro-instruction.

<number> any valid self-defining term; limits on the size of the term are described for each macro-instruction. In most cases, a predefined absolute symbol may be used.

<d(b)> any valid operand providing an *addressable* displacement and base.

<address> an addressable <name> or <d(b)>.

<nfs> a valid (and optional) name-field symbol (label) naming the macro-instruction in whose name field it appears.

[optional] square brackets around a term means that it is optional.

... an ellipsis means that the preceding item may be repeated any number of times.

Examples of these macro instructions are given below.

You may want to precede the first call to any of these macros with a

```
         PRINT NOGEN
```

statement; otherwise your listing will include many generated statements that won't have much meaning to you until you've had more experience with Assembler Language and System z.

B.1.1. The CONVERTI Macro Instruction

CONVERTI is generally used to scan a data record received by the READCARD macro instruction. It converts a string of optionally signed decimal characters to binary into a specified 32-bit or 64-bit general register, and sets GR1 to the address of the non-digit character at which scanning was stopped. This means that you can convert multiple values from the same character string.

The CONVERTI macro instruction is written in the form

```
<nfs>    CONVERTI  <number>,<address>[,ERR=<address>][,STOP=<address>]
```

where <number> specifies a general register, and <address> is the starting address of a string of bytes in storage of characters to be converted to binary. The <address> operand may also be written as <d(b)>.

The first non-blank character must be a plus sign, a minus sign, or a decimal digit; otherwise, GR1 is set to the address of that character and the register specified by <number> is unchanged. (If you expect unusual characters to be scanned, you should specify the STOP= operand on the CONVERTI statement.)

Be Careful!

Don't specify either 1 or 17 for the <number> operand, because any converted value in GR1 or GG1 will be replaced by the address of the “stop” character.

The optional keyword operands ERR= and STOP= specify locations in your program where the CONVERTI will transfer control if certain conditions occur:

ERR= If the value of the <number> operand is greater than 31, or if the value of the significant decimal digits at the <address> operand is too large to be converted correctly to the general register specified by the <number> operand, control will be transferred to the <address> given by the ERR= operand.

STOP= If an invalid character is found in the string of characters starting at the <address> operand, GR1 is set to the address of that character and control will be transferred to the <address> given by the STOP= operand. (See example 3 below.)

If either of these conditions occurs and the needed ERR= or STOP= <address> is not specified, CONVERTI will print a message and terminate the program.

32-bit general register
A <number> between 0 and 15 specifies the corresponding 32-bit general register. The number to be converted must have no more than 10 significant digits. Insignificant leading zero digits are ignored.

64-bit general register
A <number> between 16 and 31 specifies the corresponding (less 16) 64-bit general register. The number to be converted must have no more than 19 significant digits. Insignificant leading zero digits are ignored.

Execution on z/Architecture is required.

For example:

1. Convert the digits at **D1** to binary in 32-bit GR3:

```
         CONVERTI  3,D1
         - - -
D1       DC    C'  +019 '
```

The contents of GR3 will be X'00000013' and GR1 will contain the address of the blank character following the digit 9.

2. Convert the digits at **D2** to binary in 64-bit GG5:

```
         CONVERTI  21,D2
         - - -
D2       DC    C'-9223372036854775808   '
```

The contents of GG5 will be X'8000000 00000000' and GR1 will contain the address of the blank character following the final digit 8.

3. The character string at **D3** contains several values to be stored at **Table**. The scan is terminated by the character *.

```
         LA    3,Table
         LA    2,D3                    Start of string
CvtLoop  CONVERTI  0,0(2),STOP=Check   Invalid character? Test at Check
         ST    0,0(,3)                 Store an entry at Table
         LA    3,4(,3)                 Point to next Table entry
         B     CvtLoop                 Resume converting
         - - -
Check    CLI   0(1),C'*'               Is the invalid character ours?
         JNE   BadChar                 No, a bad character appeared
         - - -
D3       DC    C' +1 -2+3 -0000000000000004  *'
```

and the four words starting at **Table** will contain the values 1, −2, 3, and −4.

Providing a known "stop" character lets you scan an input string for all the values provided. For example, a record stored at **InRec** by the READCARD macro could be followed by a stop character:

```
InRec    DS    CL80                    Buffer area
StopChar DC    C'*'                    Stop character
```

B.1.2. The CONVERTO Macro Instruction

The CONVERTO macro instruction is written in the form

```
<nfs>    CONVERTO  <number>,<address>
```

where the <number> is the number of a general or floating-point register, and <address> points to a string of bytes in storage where the converted result is stored. The <address> operand may also be written as <d(b)>.

CONVERTO converts the contents of the designated register from binary to decimal characters (for general registers) or to hexadecimal characters (for floating-point registers). The first character of the converted result is always blank, so it may be printed immediately using the PRINTLIN macro.

If the contents of a general register argument is negative, a minus sign is placed before the first decimal digit.

Note that the length of the generated character string is always as specified below; be careful to allocate enough space for the result so that you won't overwrite other data or instructions.

If the value of the <number> operand does not lie in the range 0 ≤ <number> ≤ 47, the macro call is ignored.

32-bit general register

A <number> between 0 and 15 specifies the corresponding 32-bit general register. For example, if the operand is 9, the contents of GR9 will be converted to a string of 12 characters. For example, if c(GR9)=X'80000000', the formatted string will be the 12 characters (where • is our representation for a blank character):

•-2147483648

64-bit general register

A <number> between 16 and 31 specifies the corresponding (less 16) 64-bit general register. For example, if the operand is 16, the contents of GG0 will be converted to a string of 21 characters. For example, if c(GG9)=X'80000000 00000000', the formatted string will be the 21 characters (where • is our representation for a blank character):

•-9223372036854775808

Execution on z/Architecture is required.

Floating-Point Register

A <number> between 32 and 47 specifies the corresponding (less 32) Floating-Point Register.[317] For example, if the operand is 36, the contents of FPR4 will be printed as a string of 20 characters (where • is our representation for a blank character):

```
•X'FEDCBA9876543210'
```

Because there are three floating-point representations, and because accurate conversion from hexadecimal and binary floating-point formats is quite difficult, only hexadecimal values are displayed.

B.1.3. The DUMPOUT Macro Instruction

The DUMPOUT macro instruction is written in the form

```
<nfs>    DUMPOUT   <name>[,<name>]
```

where either <name> operand can be written as <d(b)>.

DUMPOUT prints a formatted hexadecimal dump of the area of memory starting with the fullword containing the first operand, 32 bytes to a line. If the second operand is omitted, one line will be printed. If the second operand is given, all of memory between the two addresses will be dumped. The dump starts from the lower of the two addresses and proceeds toward the higher. The lower address is “rounded down” to a fullword boundary, and 32 bytes are displayed on each line even if some bytes are at addresses greater than the higher address.[318]

No checks are made to avoid possible storage access violations. For example,

```
DumpA    DUMPOUT   A
```

will cause the 32-byte area of memory starting at (or very near) A to be dumped. Similarly,

```
DumpAB   DUMPOUT   A,B
```

would print a dump of the area of memory starting with a line containing the byte at A and ending with a line which includes the byte named B.

B.1.4. The PRINTLIN Macro Instruction

The PRINTLIN macro instruction is written in the form

```
<nfs>    PRINTLIN  <address>[,<number>]
```

where the <name> and optional <number> operands may also be written as <d(b)>.

PRINTLIN causes the character string beginning at the location defined by the first operand to be printed; the number of characters is specified by the second operand (which may be a predefined absolute symbol). The print-line length is limited to 121 characters.

The *first* character of the string will be *detached* and used for spacing control. The ANSI Standard carriage control characters are:

- an EBCDIC ' ' (blank) means single space,
- an EBCDIC '0' (zero) means double space,
- an EBCDIC '-' (minus) means triple space,
- an EBCDIC '1' (one) means start at the top of a new page, and
- an EBCDIC '+' (plus) means *no* spacing (the new line will be printed over the previous one).

If the second operand is omitted, the length of the character string is assumed to be 121 bytes, which means that 120 characters will be printed after the first is detached.[319] If the second operand is present, it is taken to be the length of the string; the number of characters specified will be placed at the left end of an internal buffer, extended to 121 bytes with blanks if necessary, and then sent to the printer file. For example, we could write

---

[317] If your system does not support z/Architecture instructions or the full set of 16 Floating-Point registers, trying to display their contents may cause a program interruption for an invalid operation code or a specification exception.

[318] That is, the dump is from the smaller to the larger of (LowAddr/4)×4 and ((HighAddr+31)/32)×32.

[319] See the discussion at “B.3.1. Operating System Environment and Installation Considerations” on page 1023 for information on setting the default print-line length.

```
PrTtl    PRINTLIN  Title
         - - -
Title    DC   CL121'1Title for top line of the page'
```

to print the indicated title at the top of a new page. If we wrote

```
         PRINTLIN  Title,1
```

then the printer would skip to the top of a new page and print a blank line there, because only the spacing control character (the "1") is transmitted from the program to the print file.

B.1.5. The PRINTOUT Macro Instruction

The PRINTOUT macro-instruction lets you print the value of the contents of named areas of memory, the contents of registers, and to terminate execution.

The operand field of the PRINTOUT macro-instruction may contain any number of <name>s or <number>s separated by commas, with no intervening blanks. An operand consisting of a single asterisk will terminate execution. The basic forms of the PRINTOUT macro instruction are written

```
<nfs>    PRINTOUT  [<name>...][,<number>...][,<d(b)>...]

<nfs>    PRINTOUT  *
```

where any combination of the <name> and <number> operands may be used in an operand list. Either may be written in the form <d(b)>. If the asterisk operand is used, it is treated as the last operand in the list. For example,

```
AllDone  PRINTOUT  0,*
```

will display the contents of GR0 and then terminate execution.

A <number> operand with value between 0 and 47 causes the contents of a register to be printed in hex and decimal; larger values are treated as addresses. The <number> may in most cases be a predefined absolute symbol.

32-bit general register
a <number> between 0 and 15 specifies the corresponding 32-bit general register. For example, if the operand is 12, the contents of GR12 will be printed:

```
GPR  12  = X'FFFFFFF3' =          -13
```

64-bit general register
a <number> between 16 and 31 specifies the corresponding (less 16) 64-bit general register. For example, if the operand is 16, the contents of GG0 will be printed:

```
GGR   0  = X'1234567890ABCDEF' =  1311768467294899695
```

Execution on z/Architecture is required.

Floating-Point Register
a <number> between 32 and 47 specifies the corresponding (less 32) Floating-Point Register.[320] For example, if the operand is 36, the contents of FPR4 will be printed:

```
FPR   4  = X'FEDCBA9876543210'
```

Because there are three floating-point representations, and because accurate conversion from hexadecimal and binary floating-point formats is quite difficult, only hexadecimal values are displayed.

To print the contents of the "original" four System/360 floating-point registers F0, F2, F4, and F6, we could write

```
FourFPRs PRINTOUT  32,34,36,38
```

Printing the contents of any other Floating-Point Register requires those registers to be installed and available on your machine.

To print the contents of R14 and then terminate execution, we could write

```
         PRINTOUT  X'E',*
```

320 If your system does not support z/Architecture instructions or the full set of 16 Floating-Point registers, trying to display their contents may cause a program interruption for an invalid operation code or a specification exception.

To print the contents of memory areas named A, B, and C, we could write

```
         PRINTOUT  A,B,C
```

The format of the output depends on the type attribute of the symbol naming the memory area:

- Type attribute C (character) data is shown as strings of at most 100 characters.
- Type attribute F or H data are shown as signed decimal numbers.
- If you use forms like PRINTOUT d(b) and d has attributes C, F, or H, the result will be formatted as above; otherwise, the data is displayed as 16 hexadecimal digits.
- Other type attributes cause data to be displayed as 2 to 100 hexadecimal digits, depending on the length attribute of the operand.

Specifying PRINTOUT with no operand is useful for flow tracing; only a header line is printed. An example is shown below.

```
         PRINTOUT
```

If you want a comment field on the statement, put a single comma as the operand:

```
         PRINTOUT  ,         Your comments here
```

Finally, if you want to terminate execution with no message:

```
         PRINTOUT  *,Header=NO    Terminate quietly
```

Any value other than “No” (in any mixture of upper and lower case) is treated as “Yes”.

B.1.6. The READCARD Macro Instruction

READCARD reads records into an 80-byte buffer in the program. This macro-instruction is written

```
<nfs>    READCARD  <name>[,<name>]
```

where either <name> operand may also be written as <d(b)>.

It reads a record from the input file into the 80-byte area beginning at the first operand address. If no record is available, then (1) control is returned to the instruction specified by the second operand if it is present, or (2) if no second operand is present, execution is terminated with the message

```
 *** Execution terminated by Reader EOF
```

For example,

```
         READCARD  MyRecord
```

will read the next record and place it as an 80-byte EBCDIC character string at the location named MyRecord; if no record is present, execution will be terminated. The instruction

```
GetARec  READCARD  MyRecord,EndFile
         - - -
EndFile  - - -     Do something about no more records
```

does the same as the previous example, except that if no record is available, control will be transferred to the instruction named EndFile.

B.1.7. PRINTOUT and DUMPOUT Header

Normally, the output produced by the PRINTOUT and DUMPOUT macros will be preceded by a “header” line:

```
   *** PRINTOUT requested at Address xxxxxx, Statement sssss, CC=n
or
   ***  DUMPOUT requested at Address xxxxxx, Statement sssss, CC=n
```

where sssss is the statement number of the macro, and CC=n shows the current Condition Code setting.

To suppress this header line, you can specify an additional operand HEADER=NO on the PRINTOUT or DUMPOUT macro. For example:

```
         PRINTOUT  A,B,C,Header=No
ABDMP    DUMPOUT   A,B,header=no
```

The default is HEADER=YES.

B.1.8. Usage Notes

1. All the macros require residence in RMODE(24) storage below the 16MB “line”, and execute in AMODE(24). The generated code is frequently self-modifying, and is not re-enterable.
2. Most operands of the form <name>, <d(b)>, <address>, and <number> are resolved in S-type address constants, so addressability is required when all macros except $$GENIO are invoked.
3. When you execute a macro, be sure that the base register used at assembly time to resolve the S-type constants has the correct address at execution time.
4. Be careful not to reference areas outside your program, as you may risk interruptions for memory protection violations.
5. If you use PRINTOUT to display named areas of memory, it uses the name's attributes for formatting the result.
6. At most eight characters of the <name> and <d(b)> operands are displayed. If you write

   ```
   PRINTOUT 00000000(3),00000000(7)
   ```

   the eight bytes addressed by registers GR3 and GR7 will be displayed in hexadecimal, but both will have the “name” 00000000.

B.2. Sample Program

Here is a small sample program that uses all of these macros. The assembly listing and its output are in the following figures.

```
         Print Nogen              Suppress expansions
IOSamp   CSect ,                  Sample Program
         Using *,15               Local base register
         SR    1,1                Clear card counter
*                                 Next statement for flow tracing
         PrintOut
Read     ReadCard CardOut,EOF     Read card until endfile
         LA    1,1(0,1)           Increment card counter
         PrintOut 1               Print the count register
         PrintLin Out,Linelen     Print a line
         ConvertI 2,CardOut       Convert a number into GR2
         ConvertO 2,OutData       Put it in printable form
         PrintLin OutData,L'OutData  Print the value
         B     Read               Go back and read again
EOF      DumpOut IOSamp,Last      Dump everything
         XGR   3,3                Set GG3 to  0
         BCTGR 3,0                Now set GG3 to -1
         PrintOut 1,19,32,*       Print GR1, GG3, FPR0, terminate
Out      DC    C'0Input Record = ″'  First part of line
CardOut  DC    CL80' ',C'″'       Card image here
LineLen  Equ   *-Out              Define line length
OutData  DS    CL12               Converted characters
Last     Equ   *                  Last byte of program
         End
```

The program uses READCARD to read two card images, keeps a count in GR1 which is displayed with PRINTOUT. The card image just read is shown with a prefix using PRINTLIN. Then, one number from each record is converted to binary using CONVERTI, and then to printable format using CONVERTO. The result of each conversion is printed using PRINTLIN.

When there are no more input records, DUMPOUT shows the entire program. Finally PRINTOUT * also displays the contents of GR1, GG3, and FPR0 before terminating the program.

The two 80-byte card-image input records look like this:

```
+123456                  * First record
-000034567890            * Second and last record
```

Here is a portion of the assembly listing, including a carriage control character on the first line:

```
0                                  1          Print Nogen             Suppress expansions
 000000               00000 001E4     2 IOSamp   CSect ,                 Sample Program
                 R:F  00000           3          Using *,15              Local base register
 000000 1B11                          4          SR    1,1               Clear card counter
                                      5 *                                Next statement for flow tracing
 000002 90EF F00C           0000C     6          PrintOut
 000024 0700                        743 Read     ReadCard CardOut,EOF    Read card until endfile
 00004A 4110 1001           00001   752          LA    1,1(0,1)          Increment card counter
 00004E 90EF F058           00058   753          PrintOut 1              Print the count register
 00007C 0700                        764          PrintLin Out,Linelen    Print a line
 00009C 0700                        772          ConvertI 2,CardOut      Convert a number into GR2
 0000BC 0700                        780          ConvertO 2,OutData      Put it in printable form
 0000DC 0700                        788          PrintLin OutData,L'OutData  Print the value
 0000FC 47F0 F026           00026   796          B     Read              Go back and read again
 000100 0700                        797 EOF      DumpOut IOSamp,Last     Dump everything
 000128 B982 0033                   806          XGR   3,3               Set GG3 to  0
 00012C B946 0030                   807          BCTGR 3,0               Now set GG3 to -1
 000130 0700                        808          PrintOut 1,19,32,*      Print GR1, GG3, FPR0, terminate
 000176 F0C99597A4A340D9            823 Out      DC    C'0Input Record = ″'  First part of line
 000187 4040404040404040            824 CardOut  DC    CL80' ',C'″'      Card image here
                      00062         825 LineLen  Equ   *-Out             Define line length
 0001D8                             826 OutData  DS    CL12              Converted characters
                      001E4         827 Last     Equ   *                 Last byte of program
                                    828          End
```

The output from this sample program is shown below. The listing shows the carriage control characters.

```
  *** PRINTOUT requested at Address 01A002, Statement     6, CC=0
  *** PRINTOUT requested at Address 01A04E, Statement   753, CC=0
  GPR   1  = X'00000001' =           1
 0Input Record = ″   +123456                  * First record                              ″
      123456
  *** PRINTOUT requested at Address 01A04E, Statement   753, CC=0
  GPR   1  = X'0001A192' =      106898
 0Input Record = ″   -000034567890            * Second and last record                    ″
    -34567890
  ***  DUMPOUT requested at Address 01A102, Statement   797, CC=0
  01A000  1B1190EF F00C58F0 F01405EF 00F12802 0001A000 0001A204 F001A002 00000006  *....0..00....1........s.0.......*
  01A020  98EFE000 070090EF F03058F0 F03805EF 00F12802 0001A000 8001A22C 0001A026  *q.......0..00....1........s.....*
  01A040  F18798EF E00047F0 F1024110 100190EF F05858F0 F06005EF 00F12802 0001A000  *1gq....01.......0..00-...1......*
  01A060  0001A204 0001A04E 000002F1 A0070001 F1404040 40404040 98EFE000 070090EF  *..s....+...1....1       q.......*
  01A080  F08858F0 F09005EF 00F12802 0001A000 0001A1F8 F1760062 98EFE000 070090EF  *0h.00....1.........81...q.......*
  01A0A0  F0A858F0 F0B005EF 00F12802 0001A000 0001A220 0002F187 98EFE000 070090EF  *0y.00....1........s...1gq.......*
  01A0C0  F0C858F0 F0D005EF 00F12802 0001A000 0001A214 0002F1D8 98EFE000 070090EF  *0H.00....1........s...1Qq.......*
  01A0E0  F0E858F0 F0F005EF 00F12802 0001A000 0001A1F8 F1D8000C 98EFE000 47F0F026  *0Y.000...1.........81Q..q....00.*
  01A100  070090EF F10C58F0 F11405EF 00F12802 0001A000 0001A1E8 0001A102 0000031D  *....1..01....1.........Y........*
  01A120  F000F1E4 98EFE000 B9820033 B9460030 070090EF F13C58F0 F14405EF 00000000  *0.1Uq....b..........1..01.......*
  01A140  00000000 0001A204 0001A132 00000328 20070001 F1404040 40404040 20070013  *......s.............1       ....*
  01A160  F1F94040 40404040 20070020 F3F24040 40404040 0800F0C9 9597A4A3 40D98583  *19      ....32      ..0Input Rec*
  01A180  96998440 7E407F40 404060F0 F0F0F0F3 F4F5F6F7 F8F9F040 40404040 40404040  *ord = ″   -000034567890         *
  01A1A0  40404040 5C40E285 83969584 40819584 409381A2 A3409985 83969984 40404040  *    * Second and last record    *
  01A1C0  40404040 40404040 40404040 40404040 40404040 4040407F 40404060 F3F4F5F6  *                          ″   -3456*
  01A1E0  F7F8F9F0 00000000 900FF098 9201F136 41C00008 A7F40020 900FF088 41C00004  *7890......0qk.1.....x4....0h....*
  *** PRINTOUT requested at Address 01A132, Statement   808, CC=0
  GPR   1  = X'0001A197' =      106903
  GGR   3  = X'FFFFFFFFFFFFFFFF' =                   -1
  FPR   0  = X'0000000000000000'
  *** Execution terminated by PRINTOUT * at Address 01A132
```

B.3. The Macro Instruction Definitions

The macro definitions that follow can be used to implement the macro instructions described above. An important feature of these macro instructions is that they may be used *anywhere* in a program: they make no changes to any register (except for CONVERTI, which changes the contents of GR1 and the specified result register), do not change the Condition Code, and do not contain any USING or DROP instructions that could affect addressability in your program.

The last macro definition (for `$$GENIO`) generates the module that performs the functions requested by the other macros. It does *not* require the user to do anything special about addressability, so the macros may be used in any program.

B.3.1. Operating System Environment and Installation Considerations

First, place the macro definitions in a macro library accessible to the Assembler. The default print-line length (121) can be changed in the PRINTLIN macro, and in the $$GENIO macro by modifying the variable &$$PLL. The default DDnames are

```
Print   MVS/CMS=SYSPRINT,  VSE=SYSLST
Read    MVS/CMS=SYSIN,     VSE=SYSIPT
```

and can be changed by modifying the &$$ONAM and &$$INAM variable symbols in macro $$GENIO.

The $$GENIO macro is complex. It generates the $$IOSECT CSECT with six entry points. It can be used in two ways:

1. The instructions in the $$IOSECT can be generated as part of the user's program, if the variable symbol &$$LIBIO is set to 0 in the first few lines of the $$GENIO macro. This is simpler, but causes an “invisible gap” in the statement numbers of the listing. The hidden statements can be displayed by specifying the Assembler option PCONTROL(GEN,ON) but the generated code will be confusing to all but advanced students.
2. Alternatively, you can generate the $$IOSECT instructions into a separate module. (This has the advantage of hiding the complexities of the $$GENIO macro, but requires a little bit more initial setup.) First, set the &$$LIBIO variable symbol to 0, and create and assemble this short program:

   ```
   $$GENIO
   End
   ```

   Link the generated object module into a library accessible at program linking and loading time. Then, change the $$LIBIO variable symbol to 1 to suppress subsequent inline generation, and store the macro back in the macro library.

All symbols generated in the expansions of these macros begin with the two characters $$. If this conflicts with your conventions (or desires), change each occurrence of '$$' to whatever two other characters you like. (The symbol cross-reference for any assembly using these macros will include many symbols starting with those two characters.)

The macros have been extensively tested under MVS, CMS, and VSE, and are set up to run under MVS or CMS as the default. To run them under VSE, change the &$$DOS SETB statement (near the front of the $$GENIO macro definition) according to the instructions there. Similarly, to change the default file names or print line length, modify the specifying SETC statements, as indicated there.

B.4.1. CONVERTI Macro Definition

```
         Macro
&L       CONVERTI  &R,&A,&ERR=,&STOP=
.****************************************************************
.* This macro converts a character string starting at the      *
.* address to a binary integer in the register operand. If the *
.* register operand is greater than 31, and if the number being*
.* scanned is too large, the ERR= exit is taken, and R1 is set *
.* to the address of the next character to be scanned. If an   *
.* invalid character is encountered, the STOP= exit is taken,  *
.* and R1 is set to the address of that character.             *
.*                                                             *
.* If either of these error occurs and no exit address is      *
.* provided, a message is issued and the program halts.        *
.*                                                             *
.* Generated parameter list:                                   *
.*       DC    2F'0'             For caller's R14,R15          *
.*       DC    AL1(Flag),VL3($$CNVRTI)                         *
.*       DC    S(register),S(memory_address)                   *
.*       BC    0,&ERR+0   if register or number is invalid     *
.*       BC    0,&STOP+0  if an invalid character is found     *
.* These two BCs are not generated if no keywords are present. *
.****************************************************************
         LCLA  &F
         AIF   (N'&SysList eq 2).A
         MNote 8,'CONVERTI: 2 operands required'
         MExit ,
.A       ANop  ,
&L       CNop  2,4
         STM   14,15,*+10
         L     15,*+14
         BALR  14,15
         AIF   ('&ERR' eq '').B
&F       SetA  1
.B       AIF   ('&STOP' eq '').GenData
&F       SetA  &F+2
.GenData DC    2F'0',AL1(&F),VL3($$CNVRTI),S(&R),S(&A)
         LM    14,15,0(14)
         AIF   (NOT &F).GenIO
         BC    0,&ERR+0          Error in register or number
         BC    0,&STOP+0         Invalid character found in scan
.GenIO   $$GENIO
         MEnd
```

B.4.2. CONVERTO Macro Definition

```
         Macro
&L       CONVERTO  &R,&A
.****************************************************************
.* This macro converts the value in the register operand to    *
.* a signed string of decimal characters, placing the result   *
.* in the second operand field. If the register operand is     *
.* between 32 and 47, the floating point register is converted *
.* to hexadecimal characters.                                  *
.*                                                             *
.* Generated parameter list:                                   *
.*       DC    2F'0'             For caller's R14,R15          *
.*       DC    V($$CNVRTO)                                     *
.*       DC    S(register),S(memory_address)                   *
.****************************************************************
         AIF   (N'&SysList eq 2).CVT0
.BadArg  MNote 8,'Invalid CONVERTO argument list.'
         MExit ,
.CVT0    ANOP
&V       SetA  &R
         AIF   (0 LE &V and &V LT 48).CVT1
.BadReg  MNote 8,'Invalid CONVERTO register argument.'
         MExit ,
.CVT1    AIF   ('&L' eq '').CVT2
&L       DC    0H'0'
.CVT2    CNop  2,4
         STM   14,15,*+10
```

```
         L     15,*+14
         BALR  14,15
         DC    2F'0',V($$CNVRTO),S(&R),S(&A)
         LM    14,15,0(14)
         $$GENIO
         MEnd
```

B.4.3. DUMPOUT Macro Definition

```
         MACRO
&LABEL   DUMPOUT &LOW,&HIGH,&HEADER=YES
.*****************************************************************
.* This macro dumps out an area of memory between the addresses*
.* specified by &LOW and &HIGH. The dump is in standard form,  *
.* with eight words per line along with their EBCDIC character *
.* format at the right end. The header, giving the location    *
.* from which the macro was called, is optional.               *
.*                                                             *
.* Generated parameter list:                                   *
.*       DC    2F'0'              For caller's R41,R15         *
.*       DC    V($$DMPOUT)                                     *
.*       DC    AL1(flags),AL3(call_address)                    *
.*       DC    A(calling statement number                      *
.*       DC    S(low_dump_addr),S(high_dump_addr)              *
.*****************************************************************
         LCLA   &HDR,&N           &HDR = 1 IF NO HEADER
         LCLC   &HDRC             HEADER
&N       SetA  &SYSSTMT-1         Save statement number
.NL      AIF   ('&LOW' NE '').LOW  FIRST PARM IS THERE
         MNOTE 8,'No starting address. DUMPOUT ignored.'
         MEXIT
.LOW     CNOP  2,4                ALIGNMENT
&LABEL   STM   14,15,*+10         SAVE R14 & R15
         L     15,*+14            ADDR DUMPOUT ROUTINE
         BALR  14,15              CALL
&HDRC    SetC  Upper('&HEADER')   Force to upper case
         AIF   ('&HDRC' EQ 'YES').NHDR SKIP IF A HEADER
         AIF   ('&HDRC' NE 'NO').NHDR SKIP IF A HEADER
&HDR     SETA  1                  SET TO IGNORE HEADER
.NHDR    DC    2F'0',V($$DMPOUT),AL1(&HDR),AL3(*-23),A(&N)
         AIF   ('&HIGH' EQ '').NOHIGH   NO HIGH PARM
         DC    S(&LOW),S(&HIGH) DUMPOUT ADDRESSES
         AGO   .FINIS
.NOHIGH  DC    2S(&LOW)             DUMPOUT ADDRESS
.FINIS   LM    14,15,0(14)        RESTORE REGISTERS
         $$GENIO                  GENERATE I/O SECTION
         MEND
```

B.4.4. PRINTLIN Macro Definition

```
         MACRO
&LABEL   PRINTLIN &DATA,&LEN
.****************************************************************
.* This macro sends a line image to a printer. The first       *
.* character of the string starting at &DATA is assumed to be  *
.* an ASA carriage control character. The default line length  *
.* is 121 characters, but this value may be overridden by the  *
.* value given in the optional second parameter, &LEN. If the  *
.* value of &LEN is greater than 121, then 121 will be used.   *
.*                                                             *
.* Format of parameter list:                                   *
.*       DC    2F'0'             Save R14 & R15                *
.*       DC    V($$PRTLIN),S(data),S(Len)                      *
.****************************************************************
         AIF   ('&LABEL'EQ '').AA SKIP LABEL DEFINITION
&LABEL   DC    0H'0'             DECLARE LABEL
.AA      AIF   ('&DATA' NE '').PARMOK   ERR IF NO DATA AREA
         MNOTE 8,'Missing data area parameter. PRINTLIN ignored.'
         MEXIT
.PARMOK  CNOP  2,4               ALIGNMENT
         STM   14,15,*+10        SAVE R14 & R15
         L     15,*+14           ADDR PRINTLIN ROUTINE
         BALR  14,15             CALL
         AIF   ('&LEN' EQ '').DEFLEN   IF LEN OMIT. USE DEFAULT
         DC    2F'0',V($$PRTLIN),S(&DATA),S(&LEN)
         AGO   .LM
.DEFLEN  DC    2F'0',V($$PRTLIN),S(&DATA),S(121) DEFAULT LENGTH
.LM      LM    14,15,0(14)       RESTORE REGS.
         $$GENIO                 GENERATE I/O SECTION
         MEND
```

B.4.5. PRINTOUT Macro Definition

```
       MACRO
&LABEL PRINTOUT &HEADER=YES
.****************************************************************
.* This macro lets you print register contents, areas of       *
.* memory, and terminate execution. To print an area of memory,*
.* specify its symbolic name.                                  *
.*                                                             *
.* Memory operands must be addressable by an S-type address    *
.* constant to be printed by this macro -- all such arguments  *
.* must be addressable. There may be any number of arguments   *
.* for each macro call.                                        *
.*                                                             *
.* The printed output will contain the name of the item and    *
.* the value of the named item. The output for each call may   *
.* be preceded by a header message, which will be omitted if   *
.* HEADER=NO is coded (after the last operand, usually).       *
.*                                                             *
.* To terminate execution of the program, code * as the last   *
.* (or only) operand.                                          *
.*                                                             *
.* Format of parameter list:                                   *
.*       DC    2F'0'             Save caller's R14/R15         *
.*       DC    V($$PRTOUT)                                     *
.*       DC    AL1(flags),AL3(call_address)                    *
.*       DC    A(call_statement_number)                        *
.*       operand-specific data is described later in the macro.*
.****************************************************************
         LCLA  &CNT,&LPCNT,&TCODE,&LENGTH,&A,&N
         LCLB  &BADOP            Missing/Ignored operand
         LCLC  &HDRC             Uppercase HEADER operand
.* &CNT is number of parameters; &LPCNT is loop counter;
.* &TCODE is operand code,
&N       SetA  &SYSSTMT-1        Calling statement number
         LCLC  &T                TYPE CODE OF PARAM.
&CNT     SETA  N'&SYSLIST        GET NO. OF ARGS.
         CNOP  2,4               ALIGNMENT
&LABEL   STM   14,15,*+10        SAVE REGS.
         L     15,*+14           ADDR PRINTOUT ROUTINE
```

```
         BALR  14,15              CALL ROUTINE
         DC    2F'0',V($$PRTOUT)  SAVE AREA, ADDR PRINTOUT
&HDRC    SetC  Upper('&HEADER')   Upper case
         AIF   ('&HDRC' NE 'NO').NHDR
&A       SETA  1                  INDICATE NO HEADER
.NHDR    AIF   (&CNT NE 0).SS1
&A       SETA  &A+240             INDICATE NO PARMS
.SS1     DC    AL1(&A),AL3(*-23),A(&N) PARAMETER FLAGS
         AIF   (&CNT EQ 0).ENDLOOP EXIT IF NO PARMS
.PARMLP  AIF   (&CNT LE &LPCNT).ENDLOOP TEST IF LOOP COMPLETE
&LPCNT   SETA  &LPCNT+1           INC. LOOP CNTR
&T       SETC  T'&SYSLIST(&LPCNT) TYPE CODE OF OPERAND
&TCODE   SETA  32                 HEX IS DEFAULT
&LENGTH  SETA  8                  DEFAULT LENGTH
         AIF   ('&SYSLIST(&LPCNT)' NE '*').SS4 BRANCH IF NOT *
         DC    X'0800'            PRINTOUT *
         AGO   .GENIO             GENERATE I/O SECTION
.SS4     AIF   ('&T' EQ 'C').CHAR   BRANCH IF OPER. CHAR.
         AIF   ('&T' EQ 'F' OR '&T' EQ 'H').DEC PRINT AS DEC
         AIF   ('&T' EQ 'J' OR '&T' EQ 'M').LOK DEFAULT ATTR.
         AIF   ('&T' EQ 'T' OR '&T' EQ 'U').LOK DEFAULT ATTR.
         AIF   ('&T' EQ 'N').LOKN  DEFAULT ATTRIBUTES
         AIF   ('&T' NE 'O').GETLEN   GET LENGTH ATTRIB.
         MNOTE *,'Omitted argument &LPCNT ignored.'
&BADOP   SETB  (1)                Indicate bad/missing operand
         AGO   .PARMLP
.LOKN    ANOP
&A       SETA  &SYSLIST(&LPCNT)   GET VALUE OF ARGUMENT
         AIF   (&A LT 48).LOK     BRANCH IF VALID
         MNOTE *,'Operand &LPCNT ignored: value (&A) too big.'
&BADOP   SETB  (1)                Indicate bad operand
         AGO   .PARMLP            SKIP THIS ARGUMENT
.DEC     ANOP                     F OR H CONSTANT
&TCODE   SETA  64                 PRINT AS DECIMAL HWORD
         AIF   ('&T' EQ 'H').GETLEN   GET LENGTH ATTRIB HWORD
&TCODE   SETA  65                 PRINT AS DECIMAL FWORD
         AGO   .GETLEN            GET LENGTH ATTRIB.
.CHAR    ANOP                     CHARACTER STRING
&TCODE   SETA  16                 TYPE IS CHARACTER
.GETLEN  ANOP
&LENGTH  SETA  L'&SYSLIST(&LPCNT)  LENGTH ATTRIBUTE
.* Max length chosen so printed data will fit on one print line
         AIF   (&LENGTH LE 50).LOK IF LENGTH < 50 O.K.
         AIF   (&LENGTH LE 100 AND '&T' EQ 'C').LOK CHAR, 100 OK.
&LENGTH  SETA 100                 MAX LENGTH FOR CHAR STR
         AIF   ('&T' EQ 'C').LOK
&LENGTH  SETA 50                  MAX LENGTH FOR HEX STR
.LOK     ANOP
         AIF   (&LPCNT NE &CNT).TCODEOK CHECK IF LAST OPERAND
&TCODE   SETA  &TCODE+128         LAST OPERAND
.TCODEOK ANOP
&LENGTH  SETA  &LENGTH-1          USE LENGTH-1 AS PARM.
.*************************************************************
.* PARAMS TO PRINTOUT--TYPE,LENGTH-1,S(ADDRESS),PRINTNAME
.* Type code: X'80' = last item    X'40' = decimal conversion
.*            X'20' = hexadecimal  X'10' = character
.*            X'08' = Prtout *     X'01' = decimal fullword
.*            All zero halfword means last operand was bad
.*************************************************************
         DC    AL1(&TCODE),AL1(&LENGTH),S(&SYSLIST(&LPCNT))
         DC    CL8'&SYSLIST(&LPCNT)'  8 Characters of print name
&BADOP   SETB  (0)                Indicate OK operand
         AGO   .PARMLP            GET NEXT OPERAND
.ENDLOOP AIF  (NOT &BADOP).DOLM   Check bad last operand
         DC   H'0'                Indicate null last operand
.DOLM    LM   14,15,0(14)         RESTORE REGISTERS
.GENIO   $$GENIO                  GENERATE I/O SECT
         MEND
```

B.4.6. READCARD Macro Definition

```
         MACRO
&LABEL   READCARD &DATA,&EOFADDR   BOTH ARGS ADDRESSABLE
.*****************************************************************
.* This macro reads a single 80-byte ″card image″ record into  *
.* the buffer provided by &DATA. If an end-of-file condition   *
.* is sensed on the input file, and the end-file parameter     *
.* &EOFADDR is present, then control will be returned to that  *
.* location. If no end-file parameter is present, the job      *
.* is terminated with an appropriate message.                  *
.*                                                              *
.* Generated argument list:                                     *
.*       DC    2F'0'              Save R14 & R15                *
.*       DC    AL1(flag),VL3($$READCD)                          *
.*       DC    A(calling_statement_number)                      *
.*       DC    S(data)                                          *
.*       BC    0,EndFile_Address                                *
.*****************************************************************
         LCLB   &EOFFLG           1 IF THERE IS EOFADDR
         LCLA   &T
.AA      AIF   ('&EOFADDR' EQ '').DECLAB TEST IF EOFADDR PRESENT
&EOFFLG  SETB 1                   EOFADDR PRESENT
.DECLAB  AIF   ('&DATA' NE '').PARMOK   SEE IF FIRST PARAM THERE
         MNote 8,'Missing data area parameter. READCARD ignored.'
         MEXIT                    ERROR EXIT FROM MACRO
.PARMOK  ANOP   ,
         CNOP   2,4               ALIGNMENT
&LABEL   STM   14,15,*+10         SAVE REGS 14 & 15
         L     15,*+14            ADDRESS READCARD ROUTINE
         BALR  14,15              CALL ROUTINE
&T       SETA  128*&EOFFLG
         DC    2F'0',AL1(&T),VL3($$READCD),A(*-22),S(&DATA)
         LM    14,15,0(14)        RESTORE R14 & R15
         AIF   (NOT(&EOFFLG)).GENIO   NO EOFFADDR, NO BC INSTR
         BC    0,&EOFADDR         COND. JUMP TO EOFADDR
.GENIO   $$GENIO                  GENERATE I/O SECTION
         MEND
```

B.4.7. $$GENIO Macro Definition

```
         MACRO
         $$GENIO
.*****************************************************************
.* This macro generates the code which implements the READCARD,*
.* PRINTLIN, DUMPOUT, and PRINTOUT macros. the OS version was  *
.* first implemented by James R. Low, and modified for DOS by  *
.* Paul M. Dantzig, students at Stanford University.           *
.* Later additions and extensions by John Ehrman.              *
.*****************************************************************
.* The following local set symbols determine various options
.* for the generated control section.
         GBLB  &$$IOFLG           1 IF IOSECT GENERATED
         LCLA  &$$PLL             PRINT LINE LEN, .GE. 121
         LCLB  &$$DOS             SET TO 1 IF SYSTEM=DOS
&$$DOS   SETB  0                  SYSTEM IS OS/360 et seq.
         LCLB  &$$LIBIO           1 IF IOSECT is in a library.
&$$LIBIO SetB  0                  Generate IOSect inline if 0,
.*       else generated code is in a runtime library if 1
         LCLC  &$$INAM            INPUT DDNAME
         LCLC  &$$ONAM            OUTPUT DDNAME
         LCLC  &$$CSNam,&$$CSTyp  Caller's CSect name and type
.*
         AIF   (&$$IOFLG).MExit   Exit if not required
&$$CSNam SetC  '&SYSECT'          Save caller's CSect name
&$$CSTyp SetC  '&SYSSTYP'         Save caller's CSect type
&$$IOFLG SetB  1                  Set expansion not needed flag
         AIF   (Not &$$LIBIO).Gen If not in library, gen
.MExit   MExit
.*****************************************************************
.* Register usage: R13 = local base                             *
.* R14,R15,R0,R1 = scratch and OS linkage                       *
.* R7  = local link register, R12 = call type                   *
```

```
.* R11 = parm ptr, retaddr, R10 = parm ptr(original R14)       *
.* R9  = data length, R2,R3,R4,R8 = work registers              *
.****************************************************************
.Gen     Push  Print,NoPrint      Save PRINT status
         Print OFF,NoPrint        Suppress this stuff
&$$IOFLG SETB  1                  Set flag for $$GENIO generated
.* SET OPTIONAL VALUES
&$$PLL   SETA  121                LINE LENGTH = 121
.* If the line length defined above is changed from 121 to 133,
.* remember to make the corresponding changes in the PRINTLIN
.* macro definition.
         AIF   (&$$DOS).OSNAME    GO DO DOS FILENAMES
&$$INAM  SETC  'SYSIN'            INPUT DDNAME
&$$ONAM  SETC  'SYSPRINT'         OUTPUT DDNAME
         AGO   .CSECT             GO GENERATE CSECTNAME
.OSNAME  ANOP
&$$ONAM  SETC  'SYSLST'           DOS DEFAULT OUTPUT FILE
&$$INAM  SETC  'SYSIPT'           DOS DEFAULT INPUT FILE
.CSECT   ANOP
$$IOSECT CSECT
$$IOSECT AMode 24
$$IOSECT RMode 24
         ENTRY $$READCD,$$PRTLIN,$$PRTOUT,$$DMPOUT
         ENTRY $$CNVRTO,$$CNVRTI
.*
$$DMPOUT STM   0,15,$$REGS-*(15)  SAVE REGS. R15 AS BASE
         MVI   $$FLGS-*+4(15),1   INDICATE DUMPOUT CALL
         LA    12,8               CODE FOR DUMP/PRINTOUT
         J     $$LOAD13           BRANCH TO COMMON CODE
.*
$$PRTLIN STM   0,15,$$REGS-*(15)  SAVE REGS. R15 AS BASE
         LA    12,4               CODE INDICATES PRINTLIN
         J     $$LOAD13           BRANCH TO COMMON CODE
.*
$$PRTOUT STM   0,15,$$REGS-*(15)  SAVE REGS.
         MVI   $$FLGS-*+4(15),0   INDICATE PRINTOUT
         LA    12,8               CODE FOR DUMP/PRINTOUT
         J     $$LOAD13           BRANCH TO COMMON CODE
.*
$$CNVRTO STM   0,15,$$REGS-*(15)  SAVE REGS.
         LA    12,12              CODE FOR CONVERTO
         J     $$LOAD13           BRANCH TO COMMON CODE
.*
$$CNVRTI STM   0,15,$$REGS-*(15)  SAVE REGS.
         LA    12,16              CODE FOR CONVERTI
         J     $$LOAD13           BRANCH TO COMMON CODE
.*
$$READCD STM   0,15,$$REGS-*(15)  SAVE REGS. R15 AS BASE
         SR    12,12              CODE INDICATES READ
.*
* $$LOAD13 BALR  13,0               LOAD BASE REGISTER
.*       USING *,13               ADDRESSABILITY IMPLIED
         CNOP  0,4                ALIGNMENT
$$LOAD13 JAS   13,$$MOVE          SET BASE REG, JUMP DATA
.* The following USING statement, although a comment, is implied
.* throughout the code generated here. By using absolute
.* displacements (calculated relative to $$, whose address is in
.* R13), we can avoid having to issue another using statement
.* anywhere in the code generated for the I/O package, and
.* therefore the user can call these macros with assurance that
.* there will be no adverse effects on his code, no matter how
.* tortured it may be. Note that we go to great lengths to
.* avoid the generation of literals, also.
*        USING $$,13              IS ASSUMED
.*       USING *,13               ADDRESSABILITY IMPLIED
$$       EQU   *                  SET BASE FOR R13
         AIF   (&$$DOS).REGS      NO SAVE AREA FOR DOS
         DC    18F'0'             OS SAVE AREA
.REGS    ANOP
$$REGS   DC    16F'0'             LOCAL SAVE AREA for user's regs
$$FFF    DC    A(X'FFFFFF')       MASK USED FOR EFF ADDR
$$F000   DC    A(X'F000')         MASK TO GET BASE REG.
$$ACALL  DC    F'0'               Calling address
         AIF   (&$$DOS).NUMC      SKIP DCB EXIT IF DOS
```

```
.* DCB EXIT SETS BLKSIZE TO LRECL IF NOT SPECIFIED OTHERWISE
$$DCBXIT DC    X'85',AL3(*+3)     DCB EXIT POINTER
         OC    62(2,1),62(1)      CHECK DCBBLKSIZ
         BCR   7,14               RETURN IF NOT ZERO
         MVC   62(2,1),82(1)      ELSE SET TO LRECL
         BR    14                 COMPLETE OPEN
.NUMC    ANOP
$$CVIASt DS    F                  Digit string start address
$$CVIAEn DS    F                  Digit string end+1 address
$$RDATA  DC    D'0'               For reg conversion subroutines
$$SAVG0  DC    D'0'               To save GGR0 temporarily
$$DWORD  DC   2D'0'               USED FOR CVD,FLPTR,UNPK,CVDG
         DC    X'0'               USED FOR UNPK INTO HEX
$$CVIM32 DC    P'2147483648'      Maximum 32-bit binary magnitude
$$CVIM64 DC    P'9223372036854775808' Max 64-bit binary magnitude
$$XTemp  DS    XL4                For packing high-order digits
$$FLG2   DC    X'00'              Temp save for no-header bit
$$FLGS   DC    X'00'              PRINTOUT PARAM FLGS.
$$CVIFlg DC    X'00'    80=signed; 40=+; 20=-, 01=ERR exit
.* ALSO USED TO INDICATE DUMPOUT/PRINTOUT CALL
$$CC     DC    C', CC='           FOR PRINTOUT HEADING
$$CCV    DC    C'0'               CC VALUE
$$ST     DC    C', Statement'     Statement number text
$$GPR    DC    C'GPR'             FOR REGISTER PRINTOUT
$$PC     DC    C'*** PRINTOUT requested at Address'  MESSAGE
$$DC     DC    C' DUMP'           OVERLAY FOR ABOVE MSG
$$XQUOTE DC    C'= X'''           FOR PRINTING HEX DATA
$$CVIMC  DC    C'CONVERTI: Invalid character encountered'
$$CVIMN  DC    C'CONVERTI: Invalid register or number too large'
$$EX     DC    C'*** Execution terminated by' TERMINATION MSG
$$REOF   DC    C'Reader EOF'      READCARD EOF TERMINATION
$$PEND   DC    C'PRINTOUT *'      PRINTOUT * TERMINATION
$$ATLOC  DC    C'at Address'      Where it happened
$$LOCP   Equ   L'$$EX+L'$$PEND+3  Offset for 'AT LOCATION'
         DC    C' '               USED TO CLEAR LINE BUFF
$$OUTBUF DC    CL&$$PLL' '        LINE BUFFER
$$PAT1   DC    X'40202120'        PATTERN TO PRINT Reg #
$$PAT2   DC    X'4020202020202020202120' PATTERN TO PRINT DEC.
$$PAT3   DC    0XL21'0',2X'40',17X'20',X'2120' Pattern for GGR
$$PAT4   DC    X'402020202120'    Pattern for statement number
$$DUMPTB DC    64C'.',C' ',9C'.',C'¢.<(+|&&',10C'.',C'$*);¬-/'
         DC    9C'.',C',%_>?',10C'.',C':#@''=".abcdefghi',7C'.'
         DC    C'jklmnopqr',8C'.',C'stuvwxyz',23C'.',C'ABCDEFGHI'
         DC    7C'.',C'JKLMNOPQR',8C'.',C'STUVWXYZ',6C'.'
         DC    C'0123456789',6C'.'
$$TRTAB  DC    C'0123456789ABCDEF' Hex translation taboel
$$CVITbl DS    0XL256             Input conversion translate table
*        Codes: hex 4=Invalid, 8=blank, C=+, 10=-, 14=digit
         DC    (C' ')X'04'        Invalid chars
         DC    XL1'8'             Blank
         DC    (C'+'-C' '-1)X'04' X'41'-X'4D' invalid
         DC    XL1'C'             +
         DC    (C'-'-C'+'-1)X'04' X'4F'-X'5F' invalid
         DC    XL1'10'            -
         DC    (C'0'-C'-'-1)X'04' X'61'-X'EF' invalid
         DC    10X'14'            Digits
         DC    6X'4'              Invalid
$$RCVT   DC    XL(L'$$PAT3)'0'
$$FLGSIO DC    X'00'              IOFLGS
.* BIT 0 OF $$FLGSIO ONE IF OUTPUT FILE OPENED.
.* BIT 1 OF $$FLGSIO ONE IF INPUT FILE OPENED.
.* BIT 2 OF $$FLGSIO ONE IF INPUT FILE EOF ENCOUNTERED.
         AIF   (&$$DOS).BUF       SKIP OS MACROS IF DOS
.* The List and Execute forms of the OPEN and CLOSE macros are
.* used because they do not require addressability, as do the
.* standard forms, which make regular use of implied addresses.
$$PROPEN OPEN  ($$OUDCB,(OUTPUT)),MF=L  OPEN LIST FOR SYSPRINT
$$RDOPEN OPEN  ($$INDCB,(INPUT)),MF=L   OPEN LIST FOR SYSIN
$$PRCLOS CLOSE ($$OUDCB),MF=L     CLOSE LIST FOR SYSPRINT
$$RDCLOS CLOSE ($$INDCB),MF=L     CLOSE LIST FOR SYSIN
.* Input and output DCBs. BLKSISE might be provided on DD statement.
$$OUDCB  DCB   MACRF=PM,DSORG=PS,RECFM=FBA,EXLST=$$DCBXIT,             X
               LRECL=&$$PLL,DDNAME=&$$ONAM
$$INDCB  DCB   MACRF=GM,DSORG=PS,LRECL=80,RECFM=FB,                    X
```

```
               DDNAME=&$$INAM,EODAD=$$EOF,EXLST=$$DCBXIT
         AGO   .BLANK
.BUF     ANOP                     DO DOS DEFINITIONS
$$ADDRI  DC    A($$INDCB)         ADDRESS OF INPUT DTF
$$ADDRO  DC    A($$OUDCB)         ADDRESS OF OUTPUT DTF
.* The use of X'5b' in the following two definitions is so that
.* one can change all occurrences of '$$' to some other neutral
.* characters without violating the DOS naming conventions for
.* its open and close routines.
$$OPEN   DC    2X'5B',CL6'BOPEN'  DOS OPEN ROUTINE NAME
$$CLOSE  DC    2X'5B',CL6'BCLOSE' DOS CLOSE ROUTINE NAME
$$ASA    DC    C'CBA98765432+-10 '   VALID ASA CONTROL CHARS
$$OUDCB  DTFPR CTLCHR=ASA,WORKA=YES,IOAREA1=$$IOAOU1,                  X
               IOAREA2=$$IOAOU2,DEVADDR=&$$ONAM,BLKSIZE=&$$PLL
$$INDCB  DTFCD WORKA=YES,EOFADDR=$$EOF,IOAREA1=$$IOAIN1,               X
               IOAREA2=$$IOAIN2,BLKSIZE=80,DEVADDR=&$$INAM
$$IOAIN1 DS    CL80               INPUT BUFFER 1
$$IOAIN2 DS    CL80               INPUT BUFFER 2
$$IOAOU1 DS    CL&$$PLL           OUTPUT BUFFER 1
$$IOAOU2 DS    CL&$$PLL           OUTPUT BUFFER 2
.BLANK   ANOP
$$MOVE   MVC   $$REGS+56-$$(8,13),0(14) COPY USER'S R14 & R15
.* At this point R14 points to parameter list. R12 contains a code
.* indicating which macro was called--0 means READCARD, 4 means
.* PRINTLIN, and 8 means DUMPOUT or PRINTOUT ($$FLGS set also).
.* R10 will contain a copy of R14, the first param address.
.* R11 will point to the next param in the list.
.* $$EFADDR calculates the effective address from the halfword
.* in the right half of R2 and returns it in R0 and R2.
         LR    11,14              COPY FIRST PARAM ADDRESS
         LR    10,11              COPY FIRST PARAM ADDRESS
.*                                SAVE CALLER'S CC VALUE
         LR    0,10               COPY CALLER'S BALR 14 REG
         SLL   0,2                DROP ILC
         SRL   0,30               KEEP ONLY CC
         STC   0,$$CCV-$$(0,13)   STORE IN CC= TEXT
         OI    $$CCV-$$(13),X'F0' MAKE A CHARACTER
         MVC   $$ACALL-$$(3,13),13(11) Save PO/DO/RC call address
         LTR   12,12              CHECK FOR READCARD
         JZ    $$OPNRD            BRANCH IF READCARD to open input
         JAS   7,$$OPNOUT         OPEN PRINTER
         B     *-$$(12,13)        Branch to processing routine
         J     $$LINP             PRINTLIN
         J     $$PODO             PRINTOUT/DUMPOUT
         J     $$CVTO             CONVERTO
.*       J     $$CVTI             CONVERTI (follows immediately!)
.*--------------------------------------------------------------------
.* CONVERTI -- convert to 32 or 64 bit signed integer in GR
.*--------------------------------------------------------------------
.*State 0: validate register operand
         LH    2,14(,11)          Get memory addressing halfword
         JAS   7,$$EFADDR         Convert to an address
         LR    3,2                Pointer carried in R3
         LH    2,12(,11)          Get register addressing halfword
         JAS   7,$$EFADDR         Convert to an address
         LA    11,16(,11)         Set return address
         CHI   2,31               Test register value
         JH    $$CVIER1           Value error if reg too big
         LR    4,2                Carry register number in R4
         MVI   $$CVIFlg-$$(13),0  Initialize flags
         TM    8(14),X'03'        Are there any exits?
         JZ    $$CVISt1           If no, nothing more to do
         MVC   $$CVIFlg-$$(13),8(14) Copy byte with exit flag bits
         NI    21(14),X'0F'       Reset ERR= branch mask to zero
         NI    25(14),X'0F'       Reset STOP= branch mask to zero
.*State 1: scan for non-blank: +, -, or digit
$$CVISt1 XR    2,2                Initial state
         TRT   0(1,3),$$CVITbl-$$(13)  Scan one character
         LA    3,1(,3)            Step to next char
         LA    2,*+2-$$(2,13)     Address-4 of first branch
         BR    2                  Branch per character type
         J     $$CVIErC           Invalid character; error exit
         J     $$CVISt1           Blank; repeat initial-state scan
         J     $$CVIP             Plus
```

```
         J     $$CVIM             Minus
         J     $$CVIS1A           Digit
$$CVIP   OI    $$CVIFlg-$$(13),X'C0'  Sign found, + value
         ST    3,$$CVIASt-$$(,13) Save digit starting address
         J     $$CVISt2           Now scan for digits
$$CVIM   OI    $$CVIFlg-$$(13),X'A0'  Sign found, - value
         ST    3,$$CVIASt-$$(,13) Save digit starting address
         J     $$CVISt2           Now scan for digits
$$CVIS1A OI    $$CVIFlg-$$(13),X'C0'  Set default + sign
         LR    2,3                Copy pointer for digit start
         BCTR  2,0                Back up to digit start address
         ST    2,$$CVIASt-$$(,13) Save digit starting address
         J     $$CVISt3           Now scan for more digits
.*State 2: have a sign; scan for required digit; non-digit -> error
$$CVISt2 XR    2,2                Clear GR2 for TRT
         TRT   0(1,3),$$CVITbl-$$(13)  Scan one character
         LA    3,1(,3)            Step to next char
         LA    2,*+2-$$(2,13)     Address-4 of first branch
         BR    2                  Branch per character type
         J     $$CVIEr2           Invalid character
         J     $$CVIEr2           Blank = invalid char
         J     $$CVIEr2           +     = invalid char
         J     $$CVIEr2           -     = invalid char
         CLI   0(3),C'0'          Is next char less than C'0'?
         JL    $$CVIS4C           If yes, scan is ended
         CLI   0(3),C'9'          Is it greater than C'9'?
         JH    $$CVIS4C           If yes, scan ended, R3=A(stop char)
.*State 3:  only digits allowed; everything else terminates scan
$$CVISt3 XR    2,2                Clear GR2 for TRT
         TRT   0(1,3),$$CVITbl-$$(13)  Scan one character
         LA    3,1(,3)            Step to next char
         LA    2,*+2-$$(2,13)     Address-4 of first branch
         BR    2                  Branch per character type
         J     $$CVIErC           Invalid char ends scan
         J     $$CVISt4           Blank = non-digit char
         J     $$CVISt4           +     = non-digit char
         J     $$CVISt4           -     = non-digit char
         J     $$CVISt3           Digit = repeat state 3
.*State 4: remove leading 0s; save end addr, new start addr
$$CVISt4 BCTR  3,0                Back up to stop character
$$CVIS4C ST    3,$$CVIAEn-$$(,13) Save stop address for user's R1
         LR    0,3                Copy end address
         BCTR  0,0                Back up to last digit
         L     2,$$CVIASt-$$(,13) Get starting address
$$CVIS4A CLI   0(2),C'0'          Check for leading zero
         JNE   $$CVIS4B           Exit loop if nonzero
         CR    2,0                Is the number entirely zeros?
         JNL   $$CVIS4B           Yes, have start of valid number
         LA    2,1(,2)            Step to next digit
         J     $$CVIS4A           Repeat the scan
$$CVIS4B ST    2,$$CVIASt-$$(,13) Save significance start address
         LR    1,2                Save start addr in GR1 for packing
         SR    0,2                Last-first = (Number-1) of digits
         LR    2,0                Save L-1 for packing and moving
.*State 5:  check reg type vs. length of digit string
         CHI   2,18               More than 19 digits?
         JH    $$CVIEr1           Error if so
         CHI   4,15               Check for 32- bs 64-bit register
         JH    $$CVISt6           Go process data for 64-bit reg
         CHI   2,9                Check length of 32-bit data
         JH    $$CVIEr1           Digit string too long, >10 digits
         AHI   2,X'0070'          Include length 8 for the doubleword
         STC   2,*+5-$$(,13)      Store L1,L2 in Pack instruction
         PACK  $$DWORD-$$(8,13),0(*-*,1)   Pack up to 10 digits
         CP    $$DWORD-$$(,13),$$CVIM32-$$(,13) Check vs. max 32
         JL    $$CVIS5A           If smaller, go ahead and convert
         JH    $$CVIEr1           Error if too big
         TM    $$CVIFlg-$$(13),X'C0'  Equals max; is sign positive?
         JO    $$CVIEr1           Error if +max
         LA    0,1                Create max negative result
         SLL   0,31               Have -2**31 in R0
         J     $$CVIV32           Go store and test 32-bit value
$$CVIS5A CVB   0,$$DWORD-$$(,13)  Convert to binary
         TM    $$CVIFlg-$$(13),X'A0'  Was there a minus sign?
```

```
         JNO   $$CVIV32           Skip if +
         LCR   0,0                Make the result negative
$$CVIV32 SLL   4,2                Make a word index from reg value
         ST    0,$$REGS-$$(4,13)  Store result in user's register
         J     $$CVIRet           And return to caller
.*State 5:  Convert to 64-bit register
$$CVISt6 XC    $$DWORD-$$(13),$$DWORD-$$(13) Clear high-order 8 bytes
         STG   0,$$SAVG0-$$(,13)  Save user's GG0
         CHI   2,15               Check for 16 or more digits
         JNL   $$CVIS6B           Go do 16 to 19 digits
         AHI   2,X'0070'          Simple case; set L1,L2 for pack
         STC   2,*+5-$$(,13)      Set length fields
         PACK  $$DWORD+8-$$(,13),0(*-*,1) Pack 1-15 digits
         CVBG  0,$$DWORD-$$(,13)  Convert to 64-bit binary
         TM    $$CVIFlg-$$(13),X'20'  Was there a minus sign?
         JNO   $$CVIV64           If not, prepare to deliver result
         LCGR  0,0
         J     $$CVIV64           Go store result
$$CVIS6B LR    0,2                Copy length-1 for long number
         AHI   0,-15              Length in R4 now 0-3 (16-19 digits)
         LR    2,0                Save difference
         AHI   0,X'0030'          Add in length for pack
         STC   0,*+5-$$(,13)      Store L1,L2 in pack instructin
         PACK  $$XTemp-$$(*-*,13),0(*-*,1)  Pack 1 to 4 digits
         SRP   $$XTemp-$$(13),1,0 Shift left once to eliminate sign
         LA    1,1(1,2)           Address of remaining 15 digits
         PACK  $$DWORD+8-$$(,13),0(15,1)  Pack remaining 15 digits
         MVC   $$DWORD+5-$$(3,13),$$XTemp-$$(13) Copy 1-4 digits
         CP    $$DWORD-$$(16,13),$$CVIM64-$$(13) Check digit magnitude
         JH    $$CVIEr1           Error if too large
         JL    $$CVIS6C           If smaller, go ahead and convert
         TM    $$CVIFlg-$$(13),X'C0' Max magnitude: was there a - sign?
         JO    $$CVIEr1           No, number too large by 1 bit
         LA    0,1                Set up max negative value
         SLLG  0,0,63             Now only a high-order bit in G0
         J     $$CVIV64           Go store result
$$CVIS6C CVBG  0,$$DWORD-$$(,13)  Convert to 64-bit binary
         TM    $$CVIFlg-$$(13),X'A0'  Was there a minus sign?
         JNO   $$CVIV64           No, store result
         LCGR  0,0                Complement it
$$CVIV64 LTR   4,4                Is the user's reg zero?
         JNZ   $$CVI64L           Jump if no, simpler case
         ST    0,$$REGS-$$(,13)   Store low (GR0) half of GG0
         J     $$CVIRet           No more to do; high half is set
$$CVI64L LR    0,4                Copy register number
         SLL   4,4                Move reg number left 4 bits
         OR    4,0                Now have X'rr' in R4
         STC   4,*+5-$$(,13)      Store in LMH instruction
         LMH   *-*,*-*,$$DWORD-$$(13) Load high half of user's Greg
         LR    1,0                User's reg number now in R1
         SLL   1,2                Make a word index from it
         L     0,$$DWORD+4-$$(,13) Get low half of 64-bit result
         ST    0,$$Regs-$$(1,13)  Store low half in user's register
         LG    0,$$SAVG0-$$(,13)  Restore GG0
         J     $$CVIRet           Return to caller
.*
.* $$CVIErr BCTR  3,0                Invalid value, R3=A(stop char)
$$CVIEr1 ST    3,$$CVIAEn-$$(,13) Store stop char address
         TM    $$CVIFlg-$$(13),X'01' Is there an ERR= operand?
         JZ    $$CVIErN           If not, don't set its branch mask
         OI    21(10),X'F0'       Set ERR= return branch mask to F
         J     $$CVIRet           Return to caller's ERR= address
$$CVIErN MVC   $$OUTBUF+1-$$(L'$$CVIMN,13),$$CVIMN-$$(13)
         JAS   7,$$OPNOUT         Make sure printer is opened
         JAS   7,$$PUTLIN         Print the line
         J     $$TERM1            And terminate
.*
$$CVIEr2 BCTR  3,0                Invalid char
$$CVIErC BCTR  3,0                Invalid data, R3=A(stop char)
         ST    3,$$CVIAEn-$$(,13) Store stop char address
         TM    $$CVIFlg-$$(13),X'02' Is there a STOP= operand?
         JZ    $$CVIErX           If not, don't set its branch mask
         OI    25(10),X'F0'       Set ERR= return branch mask to F
         J     $$CVIRet           Return to caller's STOP= address
```

```
$$CVIRet L     0,$$CVIAEn-$$(,13) Get address of stop character
         ST    0,$$REGS-$$+4(,13) Store in GR1 slot
         J     $$RETURN           Return to caller
$$CVIErX MVC   $$OUTBUF+1-$$(L'$$CVIMC,13),$$CVIMC-$$(13)
         JAS   7,$$OPNOUT         Make sure printer is opened
         JAS   7,$$PUTLIN         Print the line
         J     $$TERM1            And terminate
.*-------------------------------------------------------------------
.* CONVERTO -- convert 32 or 64 bit signed integer to characters
.*-------------------------------------------------------------------
$$CVTO   LH    2,12(,11)          Get register operand
         JAS   7,$$EFAddr         Convert to effective address
         LR    9,2                Save
         CHI   2,47               Check value
         JH    $$CVTX             Exit if too big, ignore the call
         LH    2,14(,11)          Get storage address operand
         JAS   7,$$EFAddr         Convert to effective address
         CHI   9,15               Want a 4-byte GPR?
         JH    $$CVTOD            No, want either GGR or FPR
         SLL   9,28               Drop off unwanted bits
         SRL   9,26               Make index for load
         L     0,$$Regs-$$(9,13)  Get the user's register
         ST    0,$$RData-$$(,13)  Store for conversion
         JAS   7,$$CVT4           Convert to a character string
         MVC   0(L'$$Pat2,2),$$RCVT-$$(13)  Move to caller's area
         J     $$CVTX             And return
*
$$CVTOD  CHI   9,31               Want an 8-byte GPR?
         JH    $$CVTOF            No, must be a FPR
         SLL   9,28               Drop off unwanted bits
         SRL   9,24               Make a register number
         STC   9,*+5-$$(,13)      Store in STG
         STG   *-*,$$RData-$$(,13) Store high half of user's GGR
         SRL   9,2                Make a word index
         L     0,$$Regs-$$(9,13)  Get low half of user's register
         ST    0,$$RData+4-$$(,13)  Store low half for conversion
         JAS   7,$$CVT8           Convert to characters
         MVC   0(L'$$Pat3,2),$$RCVT-$$(13)  Move to caller's area
         J     $$CVTX             And return
*
$$CVTOF  MVC   0(3,2),$$XQUOTE+1-$$(13) Initialize first 3 chars
         SLL   9,28               Drop off unwanted bits
         SRL   9,24               Make a register number
         STC   9,*+5-$$(,13)      Store in STD
         STD   *-*,$$RData-$$(,13) Store user's FPR
         UNPK  3(16,2),$$RData-$$(9,13) Convert to spread hex
         UNPK  18(2,2),$$RData+7-$$(2,13) Convert to spread hex
         TR    3(16,2),$$TRTAB-240-$$(13)   Translate to EBCDIC
         MVI   19(2),C''''        Insert closing quote
*
$$CVTX   LA    11,16(,11)         Set return address
         J     $$RETURN           Return to caller
.*-------------------------------------------------------------------
.* PRINTOUT/DUMPOUT HEADER LINE
.*-------------------------------------------------------------------
$$PODO   MVC   $$FLG2-$$(1,13),12(11) Copy No-header bit
         TM    12(11),1           TEST NO-HEADER BIT
         JO    $$NOHDR            SKIP HEADER IF SET
         MVC   $$OUTBUF+1-$$(L'$$PC,13),$$PC-$$(13) HEADER MSG
         CLI   $$FLGS-$$(13),0    CHECK FOR PRINTOUT
         JE    *+10               BRANCH IF PRINTOUT
         MVC   $$OUTBUF+5-$$(5,13),$$DC-$$(13) OVERLAY WITH DUMP
         MVC   $$DWORD-$$(3,13),13(11)   MOVE CALL ADDRESS
         JAS   7,$$HEXCV          CONVERT TO HEX
         MVC   $$OUTBUF+L'$$PC+2-$$(6,13),$$DWORD-$$(13) TO LINE
         LA    1,$$OUTBUF+L'$$PC+2+6-$$(,13) Do statement number
         MVC   0(L'$$ST,1),$$ST-$$(13)  Move text
         LA    1,L'$$ST(,1)       Step output pointer
         L     0,16(,11)          Get statement number
         CVD   0,$$DWORD-$$(,13)  Convert to packed
         MVC   0(L'$$PAT4,1),$$PAT4-$$(13)  Move pattern to line
         ED    0(L'$$PAT4,1),$$DWORD+5-$$(13)  Edit statement number
         LA    1,L'$$PAT4(,1)     Step output pointer
         MVC   0(L'$$CC+1,1),$$CC-$$(13)  Move CC value
```

```
.*       MVC   $$OUTBUF+L'$$PC+8-$$(L'$$CC+1,13),$$CC-$$(13) CC VALUE
         JAS   7,$$PUTLIN         PRINT CALLFROM MESSAGE
$$NOHDR  CLI   $$FLGS-$$(13),1    CHECK FOR DUMPOUT
         JE    $$DUMP             GO PROCESS DUMP
.*--------------------------------------------------------------------
.* PRINTOUT -- First, CHECK IF NULL PARAMETER LIST
.*--------------------------------------------------------------------
         TM    12(11),X'F0'       FLAGS IF NO PARAMS
         LA    11,20(0,11)        ADDR NEXT PARM OR RET.
         JO    $$RETURN           IF NO PARMS, RETURN
$$OUTLP  MVC   $$FLGS-$$(1,13),0(11)   COPY PARM FLAGS
         OC    0(2,11),0(11)      Check for null-last indicator
         JNZ   $$STAR             Not null-last, check for *
         LA    11,2(,11)          Step over indicator
         J     $$RETURN           And return to caller
$$STAR   TM    $$FLGS-$$(13),8    SEE IF PRINTOUT *
         JNO   $$GETADD           BRANCH IF NOT PRTO *
         TM    $$FLG2-$$(13),1    PRINTOUT * and no header?
         JO    $$TERM             Yes, just terminate
         MVC   $$OUTBUF+2+L'$$EX-$$(L'$$PEND,13),$$PEND-$$(13)
         J     $$TERM             AND GO TERMINATE
$$GETADD LH    2,2(,11)           ADDR HWORD PARAM.
         JAS   7,$$EFADDR         COMPUTE EFFECTIVE ADDR
         MVC   $$OUTBUF+1-$$(L'$$GPR,13),$$GPR-$$(13) GPR MSG
         MVC   $$OUTBUF+10-$$(L'$$XQUOTE,13),$$XQUOTE-$$(13)
         LA    0,X'F'             Mask for register digit
         NR    0,2                Mask off all but 4 bits
         CVD   0,$$DWORD-$$(,13)  CONVERT REG NO TO DEC
         MVC   $$OUTBUF+4-$$(L'$$PAT1,13),$$PAT1-$$(13) SET UP
         ED    $$OUTBUF+4-$$(L'$$PAT1,13),$$DWORD+6-$$(13) REGNO
         CHI   2,15               SEE IF GPR
         JH    $$TSTGGR           IF NOT, TO NEXT TEST
.* KNOW WE ARE TO PRINT CONTENTS OF a 32-bit GPR
         SLL   2,2                Form word index
         LA    2,$$REGS-$$(2,13)  ADDRESS USERS REGISTER
         MVC   $$RDATA-$$(4,13),0(2) Copy user's register contents
         LA    9,3                LENGTH-1 OF DATA
         JAS   7,$$PHEX           PRINT HEX
         MVI   2(3),C'='          Put = sign in output line
         JAS   7,$$CVT4           Convert it
         MVC   3(L'$$PAT2,3),$$RCVT-$$(13)  Move result to output
         J     $$PNPUT            Output the line
$$TSTGGR CHI   2,31               SEE IF 64-bit GPR
         JH    $$TSTFLT           IF NOT, TO NEXT TEST
.* Print contents of 64-bit General Register
         MVI   $$OUTBUF+2-$$(13),C'G'   Set msg to 'GGR'
         LA    0,X'F'             Set mask
         NR    2,0                Clear high-order bits
         SLL   2,4                Shift register number left
         STC   2,*+5-$$(,13)      Store in next instruction
         STG   *-*,$$RDATA-$$(,13)  Store GGRn high half
         SRL   2,2                Reposition register number
         LA    2,$$REGS-$$(2,13)  Point to user's low half
         MVC   $$RDATA+4-$$(4,13),0(2)  Move user's low order half
         LA    2,$$RDATA-$$(,13)  Data to convert
         LA    9,7                Length-1
         JAS   7,$$PHEX           PRINT HEX
         MVI   2(3),C'='          Put = sign in output line
         JAS   7,$$CVT8
         MVC   4(L'$$PAT3,3),$$RCVT-$$(13)   Move to output
         J     $$PNPUT            Output the data
$$TSTFLT CHI   2,47               SEE IF FLPTR
         JH    $$ISSYM            IF NOT IS SYMBOL
.* KNOW IS FLOATING PT REG
         MVI   $$OUTBUF+1-$$(13),C'F'   SET MSG TO 'FPR'
         SLL   2,4                PREPARE FOR STD
         STC   2,*+5-$$(,13)      SELECT FLPTR
         STD   *-*,$$DWORD-$$(0,13) GET CONTENTS FLPTR
         LA    2,$$DWORD-$$(,13)  ADDR CONTENTS FOR PHEX
         LA    9,7                LENGTH-1 OF DATA
         JAS   7,$$PHEX           PRINT HEX
         J     $$PNPUT            Output the data
$$ISSYM  MVC   $$OUTBUF+1-$$(8,13),4(11)   SYMBOL NAME TO BUFF.
         TM    $$FLGS-$$(13),64   SEE IF DECIMAL
```

```
         JO    $$DEC              BRANCH IF DECIMAL
         SR    9,9                PREPARE FOR IC
         IC    9,1(,11)           GET LENGTH-1
         TM    $$FLGS-$$(13),32   TEST FOR HEX
         JNO   $$SYM2             BRANCH IF NOT HEX
         JAS   7,$$PHEX           Convert the data
         J     $$PNPUT            Output the data
$$SYM2   DC    0H
         MVI   $$OUTBUF+12-$$(13),C'C'   SET FOR CHARACTERS
         STC   9,*+5-$$(,13)      STORE LENGTH INTO MVC
         MVC   $$OUTBUF+14-$$(*-*,13),0(2) MOVE CHARACTER DATA
         LA    3,$$OUTBUF+15-$$(9,13)  Address of trailing quote
         MVI   0(3),C''''         Put trailing quote
         J     $$PNPUT            Output the data
.* DECIMAL FULLWORD OR HALFWORD
$$DEC    CLI   1(11),7            Check for FD data type
         JE    $$DECD             Branch if yes, do long conversion
         LH    3,0(0,2)           GET HALFWORD
         TM    $$FLGS-$$(13),1    SEE IF WANTED FULLWORD
         JNO   *+8                IF NOT DON'T LOAD IT
         L     3,0(,2)            REALLY WANTED FULLWORD
         ST    3,$$RDATA-$$(,13)  Store for conversion
         JAS   7,$$CVT4           Convert to characters
         MVC   $$OutBuf+11-$$(L'$$Pat2,13),$$RCVT-$$(13)
         J     $$PNPUT            PRINT LINE
$$DECD   MVC   $$RDATA-$$(8,13),0(2)  Get the doubleword
         JAS   7,$$CVT8           Convert to characters
         MVC   $$OutBuf+12-$$(L'$$Pat3,13),$$RCVT-$$(13)
$$PNPUT  LA    11,12(,11)         POINT TO NEXT PARAMETER
$$PUT    JAS   7,$$PUTLIN         PRINT LINE
         TM    $$FLGS-$$(13),128  SEE IF LAST PARAMETER
         JNO   $$OUTLP            LOOP IF NOT
$$RETURN STM   10,11,$$REGS+56-$$(13)   STORE PARM,RETURN ADDRS.
         SPM   10                 RESET CALLER'S COND CODE
         LM    0,15,$$REGS-$$(13) RESTORE REGS
.* At this point all but R14 & R15 of user are restored;
.* R14 contains addr of parm list following the BALR, and
.* R15 contains return addr.
         BR    15                 RETURN TO USER
.*------------------------------------------------------------------
.* DUMPOUT
.*------------------------------------------------------------------
$$DUMP   LH    2,22(,11)          GET SECOND OPERAND
         JAS   7,$$EFADDR         SECOND EFFECTIVE ADDRESS
         LR    9,2                SAVE FOR A WHILE
         LH    2,20(,11)          GET FIRST OPERAND
         JAS   7,$$EFADDR         FIRST EFFECTIVE ADDRESS
         LA    11,24(,11)         SET RETURN ADDRESS NOW
         CR    2,9                COMPARE START TO END
         JNH   *+10               SKIP SWAP IF OKAY
         LR    0,9                SWAP HIGH AND LOW BOUNDS
         LR    9,2                2 HAS LOWER BOUND
         LR    2,0                AND R9 HAS UPPER BOUND
         LA    8,4                SET INCREMENT
         LCR   1,8                COMPLEMENT FOR MASKING
         NR    2,1                FORCE TO WORD BOUNDARY
$$DUMPA  ST    2,$$DWORD-$$(,13)  STORE LINE-START ADDRESS
         JAS   7,$$HEXCV          CONVERT TO HEX
         MVC   $$OUTBUF+1-$$(6,13),$$DWORD+2-$$(13) TO LINE
         LA    1,$$OUTBUF+9-$$(,13)      SET LINE POINTER
         MVI   $$OUTBUF+82-$$(13),C'*'  SET LEFT ASTERISK
         MVC   $$OUTBUF+83-$$(32,13),0(2) MOVE EBCDIC CHARS
         TR    $$OUTBUF+83-$$(32,13),$$DUMPTB-$$(13) XLATE
         MVI   $$OUTBUF+115-$$(13),C'*' SET RIGHT ASTERISK
         LA    0,8                SET INNER LOOP COUNT
$$DUMPB  MVC   $$DWORD-$$(4,13),0(2)   GET A WORD FROM CALLER
         JAS   7,$$HEXCV          CONVERT TO HEX
         MVC   0(8,1),$$DWORD-$$(13)   TO PRINT LINE
         AR    2,8                INCREMENT FETCH ADDRESS
         LA    1,9(,1)            AND LINE POINTER
         JCT   0,$$DUMPB          LOOP TILL LINE DONE
         JAS   7,$$PUTLIN         PRINT THE LINE
         CR    2,9                COMPARE LOWER TO UPPER
         JNH   $$DUMPA            GO WORK ON NEXT LINE
```

```
         J     $$RETURN
.*------------------------------------------------------------------
.* PRINTLIN
.*------------------------------------------------------------------
$$LINP   LH    2,14(0,11)         ADDR. HWORD USER BUFFER
         JAS   7,$$EFADDR         CALC. LINE LENGTH
         LA    4,&$$PLL           MAX LINE SIZE
         LTR   3,0                SEE IF CALC LENGTH ZERO
         JZ    *+10               IF ZERO USE LEN=MAX
         CR    3,4                SEE IF LEN GT MAX
         JNH   *+6                IF NOT USE LEN
         LR    3,4                USE LEN=MAX
         LH    2,12(,11)          ADDR HWORD. USER BUFFER
         JAS   7,$$EFADDR         CALC. EFFECT. ADDR.
         BCTR  3,0                LENGTH -1 FOR MVC
         STC   3,*+5-$$(,13)      STORE LENGTH INTO MVC
         MVC   $$OUTBUF-$$(0,13),0(2)   MOVE USER LINE TO BUFF
         MVI   $$FLGS-$$(13),128  MARK AS LAST PARAM
         LA    11,16(,11)         RETURN ADDR
         AIF   (NOT &$$DOS).GOPUT SKIP ASA CODE IF NOT DOS
         LA    2,L'$$ASA          GET LENGTH OF CHARACTERS
         SR    0,0                USED FOR USER'S CONTROL
         SR    1,1                FOR VALID CHARACTERS
         IC    0,$$OUTBUF-$$(,13) GET USER'S CONTROL CHAR
         IC    1,$$ASA-1-$$(2,13) GET A VALID ASA CHAR
         CR    1,0                COMPARE IT TO USER'S
         JE    $$PUT              GO PRINT IF OKAY
         JCT   2,*-10             INDEX DOWN BY 1 IF BAD
         MVI   $$OUTBUF-$$(13),C' '   FORCE BLANK IF BAD
.GOPUT   J     $$PUT               PRINT LINE AND RETURN
.*------------------------------------------------------------------
.* READCARD
.*------------------------------------------------------------------
$$OPNRD  TM    $$FLGSIO-$$(13),X'40'   SEE IF INPUT FILE OPEN
         JO    $$INOPN            IF SO DON'T OPEN AGAIN
         AIF   (&$$DOS).OPENRD    GO TO DOS OPEN CODE
*        OPEN  ($$INDCB,(INPUT))  OPEN INPUT FILE
         LA    1,$$RDOPEN-$$(,13) ADDR OF OPEN LIST
         OPEN  MF=(E,(1))         OPEN INPUT FILE
         AGO   .MARKRD            GO SET INPUT OPEN BIT
.OPENRD  ANOP
*        OPEN  $$INDCB            DOS OPEN MACRO
         LA    1,$$OPEN-$$(,13)   ADDR OF OPEN NAME
         LA    0,$$ADDRI-$$(,13)  ADDR OF INPUT DTF
         SVC   2                  DOS OPEN/CLOSE SVC
.MARKRD  OI    $$FLGSIO-$$(13),X'40'   INDICATE FILE OPENED
$$INOPN  LA    11,18(,11)         DETERMINE RETURN ADDR.
         TM    $$FLGSIO-$$(13),X'20'   SEE IF EOF ENCOUNTERED
         JO    $$EOFERR           IF SO ERROR
         LH    2,16(,10)          ADDR. HWORD DATA AREA
         JAS   7,$$EFADDR         CALC. EFFECTIVE ADDR.
*        GET   $$INDCB,(0)        GET NEW CARD IMAGE
         LA    1,$$INDCB-$$(,13)  ADDRESS OF INPUT DCB
         GET   (1),(0)            GET NEW CARD IMAGE
         J     $$RETURN           RETURN TO CALLER
$$EOF    TM    8(10),X'80'        SEE IF CALLER EOF EXIT
         JNO   $$EOFERR           IF NONE, ERROR
         OI    $$FLGSIO-$$(13),X'20'   MARK EOF FLAG
         OI    5(11),X'F0'        CH USER BC 0 TO BC 15
         J     $$RETURN           RETURN TO CALLER
$$EOFERR MVC   $$OUTBUF+2+L'$$EX-$$(L'$$REOF,13),$$REOF-$$(13)
.*------------------------------------------------------------------
.* TERMINATE
.*------------------------------------------------------------------
$$TERM   TM    $$FLG2-$$(13),1    Check for no message
         JO    $$TERM1            Branch if none
         MVC   $$OUTBUF+1-$$(L'$$EX,13),$$EX-$$(13) FINIS MSG
         MVC   $$OUTBUF+$$LOCP-$$(L'$$ATLOC,13),$$ATLOC-$$(13)
         MVC   $$DWORD-$$(3,13),$$ACALL-$$(13)  Get call address
         JAS   7,$$HEXCV          Convert to hex characters
         MVC   $$OUTBUF+1+$$LOCP+L'$$ATLOC-$$(6,13),$$DWORD-$$(13)
         JAS   7,$$PUTLIN         PRINT MESSAGE
         AIF   (&$$DOS).CLOSP     GO TO DOS CLOSE CODE
*        CLOSE ($$OUDCB)          CLOSE OUTPUT FILE
```

```
$$TERM1  LA    1,$$PRCLOS-$$(,13) ADDR OF CLOSE LIST
         CLOSE MF=(E,(1))         CLOSE OUTPUT FILE
         AGO   .CHKCLSR           GO TEST INPUT CLOSE
.CLOSP   ANOP
*        CLOSE $$OUDCB            CLOSE OUTPUT FILE
$$TERM1  LA    1,$$CLOSE-$$(,13)  SET ADDR OF ROUTINE NAME
         LA    0,$$ADDRO-$$(,13)  ADDR OF OUTPUT DTF
         SVC   2                  DOS OPEN/CLOSE SVC
.CHKCLSR TM    $$FLGSIO-$$(13),X'40'   CHECK IF INPUT FILE OPEN
         JNO   $$TERM2            IF NOT DON'T CLOSE
         AIF   (&$$DOS).CLOSR     GO TO DOS CLOSE CODE
*        CLOSE ($$INDCB)          CLOSE INPUT FILE
         LA    1,$$RDCLOS-$$(,13) ADDR OF CLOSE LIST
         CLOSE MF=(E,(1))         CLOSE INPUT FILE
         AGO   .TERM              GO TO TERMINATE CODE
.CLOSR   ANOP
*        CLOSE $$INDCB            DOS CLOSE MACRO
         LA    1,$$CLOSE-$$(,13)  ADDR OF CLOSE NAME
         LA    0,$$ADDRI-$$(,13)  ADDR OF INPUT DTF
         SVC   2                  DOS OPEN/CLOSE SVC
.TERM    ANOP
$$TERM2  MVI   $$FLGSIO-$$(13),0  CLEAR IO FLAGS
         AIF   (&$$DOS).EOJ       SKIP TO DOS EXIT
         SR    15,15              SET OS RETURN CODE TO 0
         SVC   3                  OS EXIT MACRO
         AGO   .PUTL              AND GO ON WITH CODE
.EOJ     EOJ
.PUTL    ANOP
.*-------------------------------------------------------------------
.* INTERNAL SUBROUTINES
.*-------------------------------------------------------------------
.* Length-1 is in R9, source address is in R2, target address in R3.
$$PHEX   LA    3,$$OUTBUF+14-$$(,13)   ADDRESS FOR HEX DIGIT
         LA    8,1(0,9)           NUMBER OF BYTES
         AR    9,8                NUMBER OF HEX DIGITS -1
         UNPK  0(3,3),0(2,2)      SPREAD HEX DIGITS
         LA    3,2(,3)            INC. LINE POINTER
         LA    2,1(,2)            INCREMENT SOURCE PTR
         JCT   8,*-14             LOOP IF MORE BYTES
         STC   9,*+5-$$(,13)      STORE LENGTH INTO TR
         TR    $$OUTBUF+14-$$(*-*,13),$$TRTAB-240-$$(13)
         LA    3,$$OUTBUF+15-$$(9,13)   ADDR. NEXT PRINT POS.
         MVI   0(3),C''''         CLOSING QUOTE MARK
         BR    7                  Return to caller
.*
$$CVT4   L     0,$$RDATA-$$(,13)  Get 32-bit binary integer
         CVD   0,$$DWORD-$$(,13)  Convert to packed decimal
         MVC   $$RCVT-$$(L'$$PAT2,13),$$PAT2-$$(13) move pattern
         LA    1,$$RCVT+L'$$PAT2-1-$$(,13) Possible sign position
         EDMK  $$RCVT-$$(L'$$PAT2,13),$$DWORD+2-$$(13) Edit it
         BNMR  7                  Return if not -
         BCTR  1,0                Back up
         MVI   0(1),C'-'          Set - sign
         BR    7                  Return
.*
$$CVT8   STG   0,$$SAVG0-$$(,13)  Save GGR0 (changed by CVDG)
         LG    0,$$RDATA-$$(,13)  Get 64-bit binary integer
         CVDG  0,$$DWORD-$$(,13)  Convert to packed decimal
         LG    0,$$SAVG0-$$(,13)  Restore user's GGR0 (used by CVDG)
         MVC   $$RCVT-$$(L'$$PAT3,13),$$PAT3-$$(13) move pattern
         LA    1,$$RCVT+L'$$PAT3-1-$$(,13) Possible sign position
         EDMK  $$RCVT-$$(L'$$PAT3,13),$$DWORD+6-$$(13) Edit it
         BNMR  7                  Return if not -
         BCTR  1,0                Back up
         MVI   0(1),C'-'          Set - sign
         BR    7                  Return
.*
$$EFADDR LA    0,X'FFF'           DISPLACEMENT MASK
         NR    0,2                DISPLACEMENT IN R0
         N     2,$$F000-$$(,13)   CALC WHICH BASE REG.
         JZ    *+16               RETURN IF BASE = 0
         SRL   2,10               BASE REG NO. AS INDEX
         AL    0,$$REGS-$$(2,13)  FORM EFFECTIVE ADDR.
         N     0,$$FFF-$$(,13)    MASK OFF HIGH-ORDER BYTE
```

```
         LR    2,0                 RESULT IN R2 ALSO
         BR    7                   RETURN
.*
$$OPNOUT TM    $$FLGSIO-$$(13),X'80'   IS OUTPUT FILE OPEN
         BCR   7,7                 RETURN NOW IF YES
         OI    $$FLGSIO-$$(13),X'80'   OUTPUT FILE BEING OPENED
         AIF   (&$$DOS).OPENP      DIFFERENT CODE FOR DOS
*        OPEN  ($$OUDCB,(OUTPUT)) OPEN OUTPUT FILE
         LA    1,$$PROPEN-$$(,13) ADDR OF OPEN LIST
         OPEN  MF=(E,(1))          OPEN OUTPUT FILE
         AGO   .CLEAR              GO FINISH OPEN
.OPENP   ANOP
*        OPEN  $$OUDCB             DOS OPEN MACRO
         LA    1,$$OPEN-$$(,13)    ADDRESS OF ROUTINE NAME
         LA    0,$$ADDRO-$$(,13)   ADDRESS OF DTF POINTER
         SVC   2                   DOS OPEN/CLOSE SVC
.CLEAR   BR    7                   RETURN TO CALLER
.*
$$PUTLIN LA    1,$$OUDCB-$$(,13)   ADDRESS OF OUTPUT DCB
         LA    0,$$OUTBUF-$$(,13) ADDRESS OF OUTPUT BUFFER
         PUT   (1),(0)             PRINT THE LINE
         MVC   $$OUTBUF-$$(&$$PLL,13),$$OUTBUF-$$-1(13) CLEAR
         BR    7                   RETURN TO CALLER
$$HEXCV  UNPK  $$DWORD-$$(9,13),$$DWORD-$$(5,13) UNPACK 4 BYTES
         TR    $$DWORD-$$(8,13),$$TRTAB-240-$$(13) TO EBCDIC
         BR    7                   RETURN TO CALLER
.*-------------------------------------------------------------------
         AIF   ('&SYSSTYP'eq '').NoSect
&$$CSnam &$$CSTyp                  RESTORE ORIGINAL SECTION
.NoSect  Pop   Print,NoPrint       Restore PRINT status
         MEnd  ,                   End of $$GENIO macro
```

Glossary of Terms and Abbreviations

```
 GGGGGGGGGG   LL
GGGGGGGGGGGG  LL
GG        GG  LL
GG            LL
GG            LL
GG            LL
GG     GGGGG  LL
GG     GGGGG  LL
GG        GG  LL
GG        GG  LL
GGGGGGGGGGGG  LLLLLLLLLLLL
 GGGGGGGGGG   LLLLLLLLLLLL
```

Special Characters

***** (1) Multiplication operator (2) Location Counter Reference.

+ Addition operator.

− Subtraction operator.

/ Division operator.

( ) (1) Address constant delimiters. (2) Expression grouping delimiters.

= (1) Literal-constant indicator. (2) Indicator of a keyword argument or parameter in a macro.

• Indicator of a blank space in this text.

' Apostrophe. (1) Character string delimiter. (2) Attribute reference operator.

_ Alphabetic character in symbols.

$ Alphabetic character in symbols; not invariant across all EBCDIC code pages.

@ Alphabetic character in symbols; not invariant across all EBCDIC code pages.

Alphabetic character in symbols; not invariant across all EBCDIC code pages.

& Ampersand; indicates the start of a variable symbol.

. (1) Qualified-symbol separator between qualifier and symbol. (2) Concatenation operator in conditional assembly SETC expressions.

A

absolute symbol
A symbol whose value behaves in expressions like a self-defining term. Its value does not change if the assumed origin of the program changes.

ACONTROL
Assembler instruction statement allowing dynamic modification of some Assembler options.

adcon
Abbreviation for "address constant".

addend
see *augend*

addr(x)
Address of some operand "x".

address
(1) (*n*) A number used by the processor at *execution time* to locate and reference operands or instructions in memory. Here, an address is what reference manuals (such as the *Principles of Operation*) would call a virtual address. Sometimes used (incorrectly) to mean an *assembly time location*. (2) (*v*) To reference; to provide an *address* (sense no. 1) that may be used to reference an item in storage.

address constant ("adcon")
A field within a control section into which a value (typically, an address) is placed during assembly, program linking, relocation, and/or loading.

address resolution
The process whereby the Assembler converts *implied addresses* into *base-displacement* form using information in its *USING Table*, or resolves offsets in relative-immediate instructions.

address table
A table of addresses of individual rows or columns of an array, allowing faster access to the elements of that row or column.

address translation ("Dynamic Address Translation", DAT)
The procedure used by the CPU to convert virtual addresses into real addresses.

addressability
(1) The ability of the Assembler to calculate a displacement and assign a base register to an implicit addressing expression, using information in the USING Table; or the ability of the Assembler to assign a valid offset for a relative-immediate reference. (2) The ability of an executed instruction to reference an intended location in memory.

addressability error
(1) Inability of the Assembler to derive a valid addressing field for an implicit operand. (2) An execution-time interruption for attempting to reference a non-existent address.

addressable
(1) At *assembly time* an *implied address* is addressable if it can be validly *resolved* by the Assembler into a *base-displacement address*, using information contained in the *USING Table* at the time of the resolution, or the Assembler can assign a valid offset to a relative-immediate instruction. (2) At *execution time* an operand is addressable if it lies either in the bytes starting at address zero, or is within the base-displacement resolution range of one of *general purpose registers* 1 through 15, or can be referenced by a relative-immediate instruction.

addressing halfword
A halfword containing a *base register specification digit* in the first 4 bits, and an unsigned *displacement* in the remaining 12 bits. A key element of System z addressing.

addressing mode
One of three modes (24, 31, and 64) supported by System z that determines the length of an Effective Address and the addressable areas of memory.

algorithm
A finite sequence of well-defined steps for solving a problem.[321]

AMODE
An abbreviation for "addressing mode".

anchor
(1) The *base location* or *base register* specified in the second operand of a USING statement. (2) The base location in a Dependent or Labeled Dependent USING statement. (3) The starting point of a chained list or queue.

AND operation
A logical (boolean) operation between two bits, whose result is 1 only if both operand bits are 1.

architecture
A description of "the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, and the physical implementation."[322]

argument
A value supplied by a calling program.

arithmetic division
Division of two signed operands, generating a signed quotient and signed remainder.

arithmetic multiplication
Multiplication of two signed operands, generating a signed product.

arithmetic representation
A signed number representation.

arithmetic shift
A movement of bits in a general register to the left or right, preserving the arithmetic sign of the operand.

array
A collection of data items of the same data types and lengths, arranged in contiguous storage locations. Usually accessed using one or more index variables or "subscripts". (See also *table*.)

ASCII
American Standard Code for Information Interchange, an 8-bit encoding.

ASCII numeric characters
ASCII characters with representations between X'30' and X'39'.

Assembler
A program that translates programs written in Assembler Language to machine language instructions and data, producing an object module.

Assembler Language
A lower-level language allowing programmers maximum freedom in specifying processor instructions, providing powerful "macro-instruction" facilities supporting encapsulation and economy of expression.

assembly time
The time when the Assembler is processing a program's statements, as distinct from the time when the machine language instructions created from an Assembler Language program are executed by the processor.

attribute
A property of a *symbol* known to the *assembler*, typically the characteristics of the item named by the symbol, such as its type, length, etc. A program may request the assembler to provide values of symbol attributes using *attribute references*.

A *variable symbol* may have one attribute specific to the symbol itself (e.g. its number attribute), and many attributes specific to the *value* of the *variable symbol*.

attribute reference
A notation used to request the value of a *symbol attribute* from the assembler's *symbol table*, or of a *variable symbol* or its value.

augend
When two numbers are added, the number being **aug**mented (the first operand) is the **aug**end, to which the **add**end (the second operand) is **add**ed.

B

B_n, b_n
Base register specification digit for machine instruction operand n.

B-tree
A tree whose nodes contain one or more data elements, and two or more links to successor nodes.

base address
(1) The *address* in one of *general purpose registers* 1 to 15 to which a *displacement* is added to obtain an

---

321 After *al Khwarizmi*, a nickname of the 9th century Persian astronomer and mathematician Abu Jafar Muhammad ibn Musa, who authored many books on arithmetic and algebra. He worked in Baghdad and his nickname alludes to his place of origin, Khwarizm (Khiva), in present-day Uzbekistan and Turkmenistan.

322 G.M. Amdahl, G.A. Blaauw, and F.P. Brooks, Jr. *Architecture of the IBM System/360*, IBM Journal of Research and Development Vol. 8 No. 2, 1964, reprinted in IBM Journal of Research and Development Vol. 44 No. 1/2, January/March 2000.

effective address. (2) The first operand of a USING statement. (3) The execution-time contents of a base register.

base digit
See *base register specification digit*.

base-displacement addressing
A technique for addressing memory using a compact *base-displacement* format for representing the derivation of storage addresses.

base location
(1) In *base-displacement address resolution*, the first operand of a USING statement, relative to which *displacements* are to be calculated. For ordinary USING statements, the base location is assumed to be at a relative offset (*displacement*) of zero from the address contained in the *base register*; for *dependent USING* statements, the base location may be at a positive or negative offset from the location specified in the *base register* eventually used to resolve an *implied address*. (2) Informally, this term is sometimes used to mean (a) the origin of a control section, (b) a *base address* in a register at *execution time*, and (c) whatever the speaker likes.

base register
(1) The second (and subsequent) operand(s) of a USING instruction. (2) A general register used at execution time to form an Effective Address.

base register specification digit
The first 4 bits of a 16- or 20-bit addressing field.

BCD
Binary Coded Decimal. (1) A 4-bit encoding of the decimal digits 0-9, used in packed decimal arithmetic. (2) A 6-bit character encoding used on many data processing systems prior to System/360's introduction of an 8-bit byte with EBCDIC encoding.

BEAR
The Breaking Event Address Register.

bias
A fixed value added to the exponent of a floating-point number so that its exponent field always contains a nonnegative value, the characteristic.

biased rounding
A common type of rounding that introduces a small inaccuracy in the rounded results; typically caused when rounding decimal values by adding 5 to the last discarded digit.

Big-Endian
A representation of numbers in which the value of the digits at successively higher addresses have *lower* significance; the digits have decreasing significance from left to right. The representation used on System z.

binary floating-point
A floating-point representation having a binary significand.

binary search
A technique for searching ordered arrays by probing the midpoint of successively smaller portions of the array.

binary tree
A data structure in which each element contains links to two other elements, a “left subtree” or “left child”, and a “right subtree” or “right child”.

bind time
A time following *assembly time* during which one or more *object modules* are combined to form an executable module, ready for loading into memory at *execution time*. Also known as “link time”.

Binder
The z/OS program that can generate load modules and program objects, as well as place a just-linked program directly into memory.

bit
A binary digit, taking values 0 and 1.

blank
A nonempty, finite-width invisible character; a space. In contexts where explicit blank spaces appear, we sometimes use the “•” character.

boundary alignment
(1) The Assembler's action in incrementing the Location Counter so that its value is adjusted to the boundary required by an instruction or by a constant operand. (2) The binder's action in ensuring proper alignment of the components of a load module or program object. (3) The Program Loader's action in ensuring that memory alignment of the components of a loaded program are correct. (4) The alignment of the starting address of storage acquired at execution time.

branch address
The address from which the next instruction will be fetched if a branch condition is met.

branch condition
The CPU's decision whether to alter the normal sequential execution of instructions by fetching instructions at the branch address.

branch mask
A 4-bit field in a Branch on Condition instruction used to test the value of the Condition Code. If a 1-bit in the branch mask matches the CC value, the branch condition is met.

byte
A group of 8 bits; the basic addressable unit of memory.

C

C(x), c(x)
Contents of something named “x”.

C-string
A string of zero or more bytes ending with a zero or “null” byte.

calling point identifier
A NOP instruction with a halfword identifying number in place of its addressing halfword, of the form `X'4700nnnn'`.

CC
Condition Code, a 2-bit field in the PSW used to indicate the status or result of executing certain instructions.

characteristic
The true exponent of a floating-point number plus the bias.

Class
(1) A cross-section of program object data with uniform format, content, function, and behavioral

attributes. (2) A component of a program object with specified loading properties, containing elements supplied by sections. Loadable classes are independently relocatable. Indicated in the External Symbol Directory listing with type ED.

code
An informal term for groups of Assembler Language statements.

code page
A defined collection of characters and control codes and their associated binary encodings.

cohort
In a given format, a set of decimal floating-point numbers having the same numeric value but different quanta.

column order
A way to store arrays so that the elements of each column follow one another in memory. For arrays of two or more dimensions, subscripts cycle most rapidly from left to right.

column-major order
Same as *column order*.

COM
An assembler instruction statement declaring the start or resumption of a *common section.*

common section
A control section having length and alignment attributes (but no text) for which space is reserved in the load module or Program Object. Common sections receive special treatment during program linking: space is allocated for the greatest length received for all common sections with a given name.

comparand
(1) A quantity whose value is being compared. (2) A quantity to which an incremented index is compared to determine whether a loop should be repeated.

complement addition
The addition of binary or packed decimal operands of unlike sign.

complement decimal addition
The addition of packed decimal operands of unlike sign.

complex relocatability
A property of a symbol or expression whose relocation attribute is neither absolute or simply relocatable.

conditional assembly
A form of assembly whose input is a mixture of *conditional assembly language* and *ordinary assembly language* statements, and whose outputs are statements of the *ordinary assembly language*. Statements of the *ordinary assembly language* are treated as character strings, and are not obeyed during conditional assembly.

conditional no-operation
An Assembler CNOP instruction that may generate NOP and NOPR instructions, causing the Location Counter to be aligned on a specified even boundary.

constant type
A letter specifying the desired internal data representation for a generated constant.

control section
An indivisible unit of instructions, data, or uninitialized space that is not further subdivided during linking and loading.

The smallest independently *relocatable* unit of instructions and/or data. All elements of a given control section maintain the same fixed relative positions to one another at *assembly time*. These fixed relative positions at *assembly time* are maintained by the program after control sections are placed into storage at *execution time*.

CPU
Central Processing Unit

CSECT
(1) An assembler instruction statement indicating the start or continuation of a *control section.* (2) An informal term for a control section

C-string
A string of zero or more bytes ending with a zero or “null” byte.

Cx
Characteristic part of a floating-point number “x”

CXD-type address constant
A word holding the length (not the address) of the virtual area created at link time from all the Dummy External Sections (PseudoRegisters) in the complete program.

D

D_n, d_n
Displacement specification for machine instruction operand n.

data exception
An interruption condition caused by invalid data.

Data Exception Code
A field in the FPCR indicating which of various floating-point or packed decimal exceptions may have occurred.

DBCS
See “Double-Byte Character Set”.

decimal exponent
A letter E attached at the end of some numeric constants, followed by a positive or negative integer giving the power of ten by which the preceding nominal value is multiplied.

decimal data exception
An exception condition caused by invalid numeric or sign digits in a packed decimal operand, or by invalid operand lengths for a packed decimal product.

decimal divide exception
An exception condition caused by a packed decimal quotient being too large for the available space in the first operand field, or by division by zero.

decimal floating-point
A floating-point representation having a decimal significand.

decimal overflow exception
An exception condition caused by a packed decimal sum or difference being too large for the receiving first operand field.

decimal specification exception
An exception condition caused by a packed decimal multiplication or division specifying incorrect lengths for one or both operands.

declet
A 10-bit encoding of three Binary Coded Decimal (BCD) digits. Declets may have two forms: (1) *canonical*: 1,000 preferred (and generated) values, and (2) *non-canonical*: any of 24 non-preferred encodings accepted as operands, but not generated by any arithmetic operation.

decode
The CPU action of analyzing the contents of the instruction register to determine the validity and type of an instruction.

defined symbol
A symbol is defined when the Assembler assigns values to its value, relocation, and length attributes.

denormalization
A process of shifting the significand of a floating-point number to the right by enough digit positions so its exponent will lie in a desired representable range.

denormalized number
(1) A floating-point number with denormalized significand. (2) A nonzero binary floating-point value with characteristic zero and a nonzero fraction.

dependent USING
A USING statement allowing implicit references to symbols in areas mapped by more than one DSECT to be resolved with a single base register, in which the first operand is based or *anchored* at a relocatable address. May also take the form of a labeled dependent USING statement. See also *anchor*, *labeled USING*, and *ordinary USING*.

destructive overlap
Destructive overlap occurs when any part of a target operand field is used for a source after data has been moved into it by the same instruction.

DH
In an instruction supporting signed 20-bit displacements, the 5th byte of the instruction containing the signed **H**igh-order 8 bits of the **D**isplacement.

digit selector
One of two edit-pattern characters: a Digit Selector (DS) having representation X'20', or a Digit Selector and Significance Starter (SS) having representation X'21'.

diminished radix-complement representation
A signed representation where the complement of a number is represented by subtracting each digit from (the radix minus 1). (See *two's complement representation.*)

displacement
(1) An unsigned 12-bit integer field of an *addressing halfword*, or a signed 20-bit integer field, both used in generating Effective Addresses. (2) Sometimes used to describe the offset (difference) between a given storage address and a *base address* that might be used to *address* it.

dividend
A number to be divided by a divisor; the first operand; the numerator.

divisor
A number to be divided into a dividend; the second operand; the denominator.

DL
In an instruction supporting signed 20-bit displacements, the unsigned **L**ow-order 12 bits of the **D**isplacement.

Double-Byte Character Set
A 16-bit (Double-Byte) EBCDIC encoding of a character set having many more characters than can be accommodated in 8 bits.

double-ended queue
Same as *queue*. Sometimes called a "dequeue".

doubly-linked list
Same as *queue*. Sometimes called a "double-threaded" list.

DPD
Densely Packed Decimal, a representation used for encoding decimal floating-point significands.

DROP instruction
An assembler instruction telling the Assembler to eliminate one or more entries from its USING Table.

DSECT
(1) An assembler instruction defining the start or continuation of a Dummy Control Section. (2) A Dummy Control Section, a template used to map the components of a data structure. (3) An informal name for a Dummy Control Section.

dummy section
(1) A control section generated by the DSECT statement containing no instructions or data. (2) A virtual control section created at bind time by the Binder; generated by the DXD instruction, or a reference to a DSECT by a Q-type address constant.

duplication factor
The number of times a constant operand should be assembled.

DXC
Data Exception Code, a field in the Floating-Point Control Register (FPCR).

DXD
An assembler instruction statement defining an *External Dummy Section*.

E

EAR (Effective Address Register)
A (conceptual) internal register used to calculate Effective Addresses.

EBCDIC
Extended Binary Code Decimal Interchange Code, an 8-bit encoding used to assign numeric values to character representations. The many EBCDIC encodings assign different values to some characters, but all the alphabetic, numeric, and other syntactic characters used in the Assembler Language are *invariant* across EBCDIC encodings, *except* for the characters "$", "@", and "#".

Effective Address
The address calculated at execution time from a 16- or 20-bit addressing field of an instruction, possibly with indexing, or the address calculated from a

relative-immediate offset and the address of its instruction.

element
A component of a program object class, defined by the combination of its section name and its Class name. The smallest indivisible and separately relocatable portion of a program object.

Encoded Length
The contents L_n of a Length Specification Byte or digit; one less than the value of the Length Expression N_n (unless the Length Expression is zero, in which case the Encoded Length is also zero).

entry point
The first instruction to receive control when a routine is invoked.

entry point identifier
A string of characters following the first instruction at an entry point, providing descriptive information about the entry.

EQU Extended Syntax
Additional operands on EQU instructions that provide additional information about the attributes of the symbol defined by the EQU statement.

ESD
External Symbol Dictionary.

Ex
Exponent part of a floating-point number “x”

exception condition
(1) An indication of an unusual result or condition. Some exceptions can deliver a default result if an interruption has been masked off by appropriate settings, while others always cause an interruption. (2) One of six conditions defined by the IEEE Floating-Point Standard: invalid operation, division by zero, exponent overflow, exponent underflow, quantum. and inexact result.

executable control section
A control section containing machine language instructions or data, defined by CSECT, RSECT, or START instructions.

execute
The CPU's action of performing the operation requested by the instruction in the instruction register.

execution time
The time when your program has been put in memory by the Program Loader and given control. This may happen long after *assembly time*.[323]

explicit address
An address in which you specify the base register specification digit and the displacement as absolute expressions.

explicit length
(1) A length value specified in a DC or DS statement. (2) A length field that you specify explicitly, rather than having the Assembler assign the length field from the Length Attribute of an operand.

exponent
The power of the radix by which the significand of a floating-point number must be multiplied to determine its value.

exponent modifier
A modifier specifying a positive or negative power of ten to be multiplied by the nominal value of certain numeric constants.

exponent overflow
A condition arising when the exponent of a calculated floating-point result is too large to be contained in its representation.

exponent underflow
A condition arising when the exponent of a calculated floating-point result is too small to be contained in its representation.

expression
A combination of terms and operators to be evaluated by the Assembler.

expression evaluation
The procedure used by the Assembler to determine the value of an expression.

extended mnemonic
An instruction mnemonic provided by the Assembler allowing you to specify a branch mask or other instruction fields implicitly.

External Dummy Section
An area having length and alignment defined at assembly time in a DXD statement that will be assigned at link/bind time to a *PseudoRegister Vector* and for which memory is allocated at execution time.

external symbol
A symbol whose name (and possibly, value) are a part of the object module text provided by the Assembler. Such names include (1) *control section* names, (2) strong external names declared in V-type address constants or EXTRN statements, (3) weak external names declared in WXTRN statements, (3) names of *common sections*, (4) names of *Pseudo Registers* or *external dummy sections*, (5) referenced names declared on ENTRY statements, (6) class names, and (7) symbols and character strings renamed through the use of the ALIAS statement. Compare to *internal symbol*.

External Symbol Dictionary
The set of external symbols created by an assembly. They are displayed in the Assembler's listing and are encoded in the ESD records of the generated object module.

extreme exponent
A floating-point number's exponent with maximum positive or negative value.

F

fetch
The CPU's action of bringing halfwords from memory into the conceptual Instruction Register to be interpreted as an instruction.

field separator
An edit-pattern character (FS) having value `X'22'` indicating that a packed decimal value from the second operand has been edited, and editing should continue with the following packed decimal value.

---

[323] Or, the time at which programmers whose programs consistently fail to execute correctly are themselves executed.

fill character
The first byte of an edit pattern.

firmware
A popular term referring to microcode or millicode.

floating-point
A data representation with a sign, an exponent, and a set of significant digits.

Floating-Point Control Register (FPCR)
A special register containing IEEE masks, status indicators, Data Exception Code, and rounding modes.

floating-point exception condition
One of five conditions defined by the IEEE Floating-Point Standard: invalid operation, division by zero, exponent overflow, exponent underflow, and inexact result.

floating-point system FP(r,p)
A floating-point data representation with a specified radix **r** and number of significant digits **p**, sometimes denoted FPF(r,p) or FPI(r,p) depending on the type of encoding.

floating-point system FPF(r,p)
A floating-point system with radix **r** and **p** digits of precision, in which the significant digits are represented as fractions.

floating-point system FPI(r,p)
A floating-point system with radix **r** and **p** digits of precision, in which the significant digits are represented as integers.

FPCR
Floating-Point Control Register

FPR
Floating-Point Register

FPRn, FR_n
Floating-Point Register n

free storage list
A list containing unused and available elements. Sometimes abbreviated *FSL*.

Fx
Fraction part of a floating-point number "x"

G

General Purpose Registers
A set of 16 64-bit registers used in the System z family of processors for addressing, arithmetic, logic, shifting, and other general purposes. Compare to other registers described in the *z/Architecture Principles of Operation* such as *Access Registers*, *Control Registers*, and *Floating Point Registers*.

GGn
A notation referring to 64-bit general register "n".

GR
General Register, General Purpose Register

GR G_n
A notation referring to the 64-bit general register specified by G_n.

GR R_n
A notation referring to the rightmost 32 bits of the general register specified by R_n.

GRn
A notation referring to the rightmost 32 bits of general register "n".

glyph
The printed or displayed form of a character that can be formed with various properties. For example, the glyphs A, *A*, **A**, ***A***, A are representations of the character "A", in upper case forms (normal, italic, bold, bold italic, and "small caps").

generalized object file format (GOFF)
An extended form of *object module* produced by High Level Assembler, providing numerous enhancements and extensions not supported by the traditional card-image *object module* format ("OBJ").

GOFF
See *generalized object file format*.

GOFF option
An *option* that causes High Level Assembler to generate an *object module* using the *generalized object file format*.

GPR
See *General Purpose Register*

gradual underflow
A technique allowing numbers to become denormalized when they are finite and smaller than the smallest normalized magnitude.

graphic data type
A representation of characters using a 16-bit encoding.

guard digit
An extra digit used to increase the accuracy of a calculated floating-point result.

H

hash function
A function that creates a randomized linear subscript from a data element. Used to avoid lengthy table searches for large or complex data items.

hash table
A table of data items (possibly including lists and pointers to other data items) whose entries are accessed using the results of a *hash function*.

hexadecimal
A base-16 representation, with digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, in increasing numerical order.[324]

hex
See *hexadecimal*.

HLASM
An acronym for the *High Level Assembler*.

High Level Assembler
IBM's most modern and powerful symbolic assembler for the System z processors, running on the

324 The base 16 representation was originally called "sexadecimal".

z/OS, z/VM, z/VSE, and zLinux operating systems.[325] The Assembler we describe here.

I

I
Single operand of SVC instruction.

I'
Integer Attribute Reference to a symbol (as in `I'SYM`) or to a symbolic parameter in a macro (as in `I'&PARAM`)

I_n
Immediate value specification for machine instruction operand n

IA
Instruction Address (z/Architecture PSW bits 64-127).

IC
Interruption Code, a value indicating the cause of an interruption.

ILC
See *Instruction Length Code*.

immediate operand
An instruction's operand contained in a field of the instruction itself.

implied address
An address to which you expect the Assembler to assign a base register specification digit and a displacement, or a relative-immediate offset, to an addressing field.

implied length
A length field completed by the Assembler based on its analysis of an operand.

increment
(1) A (normally) constant value used to update the value of an *index* for each iteration of a loop. (2) A (typically small) value added to another.

index
(1) The contents of an index register. (2) A varying quantity used to control each iteration of a loop.

index register
One of *general purpose registers* 1 through 15 specified by the *index register specification digit* in an RX-type instruction.

index register specification digit
4 bits of an RX-type instructions specifying a register with a value to be added to the Effective Address calculated from a base-displacment address.

indexing
Computation of an Effective Address by adding a displacement to the contents of a base register and an index register.

infix notation
The traditional form of writing arithmetic expressions, where operators are placed between operands, as in `2*(3+4)`.

inorder tree traversal
A technique for traversing a binary tree, visiting first the left subtree, then the parent node, and then the right subtree.

insert
Place one or more bytes into a register without changing other bytes.

Instruction Address (IA)
A 64-bit field in the PSW giving the address of the next instruction to be executed.

instruction cycle
A conceptual view of the CPU's actions in executing an instruction, visualized as occurring in the steps of fetch, decode, and execute.

Instruction Length Code
A 2-bit field in low storage indicating the length in halfwords of an instruction that caused a particular type of interruption.

Instruction Register (IR)
A conceptual internal register used by the CPU to decode instructions.

internal symbol
A symbol naming an element of an *Assembler Language* program, which is assigned a single value by the *Assembler*, and not part of the object module. They may be retained in the *SYSADATA file*. Compare to *external symbol*.

internal symbol dictionary
See *symbol table*.

interruptible
An instruction is interruptible if the CPU suspends its operation, updates the registers involved in the operation and subtracts the instruction's length from the Instruction Address in the PSW, so that when the program resumes execution, the instruction will start from the point where it was interrupted.

interruption
A process taking control away from the currently executing instruction stream, saving information about the interrupted program, and giving control to the Operating System Supervisor (which may in turn pass control to a program-specified routine).

invariant EBCDIC character
Those 82 characters whose EBCDIC representations do not change among EBCDIC code pages. The syntactic characters used by HLASM are defined on IBM Code Page 640.

IR
(1) Instruction Register, a conceptual internal register in the CPU into which fetched instructions are placed and decoded during the fetch-decode-execute cycle. (2) An internal register holding a target instruction so its second byte may be modified (if required) by an Execute instruction prior to final decoding.

[325] The name is not necessarily an oxymoron, as High Level Assembler can do much more than ordinary (low-level) assemblers.

J

Job Control Language
The statements needed to tell an Operating System how to process your program through the assembly, linking, and execution phases. "JCL" for short.

jump
An informal name for a relative branch, to distinguish it from a based branch using base-displacement Effective Addresses.

K

K'
Character count attribute reference to a conditional assembly symbolic parameter or SETC symbol (as in K'&PARAM)

L

L'
Length Attribute Reference to an ordinary symbol (as in L'SYMBOL). or conditional assembler symbolic parameter (as in L'&PARAM). See *Length Attribute Reference*.

L_n
Length specification digits for operands in instructions that support variable-length operands (See also N_n.)

label
(1) Colloquially, the name of an instruction or data definition. This is more properly called a *name field symbol*. (2) In High Level Assembler, the name field symbol of a USING statement, designating that statement as a *labeled USING*. The symbol is then defined as a *qualifier*.

Labeled Dependent USING
A USING statement allowing implicit references to symbols in areas mapped by more than one DSECT to be resolved with by a specific base register.

Labeled USING
A USING statement directing resolution of implicit references to a specific base register, distinguished from an ordinary USING statement by the presence of a qualifier symbol in the name field. Symbolic expressions resolved with respect to a labeled USING must use a *qualified symbol* with the *qualifier* of that labeled USING.

LC
Location Counter. See *Location Counter*.

Length Attribute Reference
A term whose value is the length attribute of a symbol. The length-attribute operator is L'.

Length Expression
A length value (denoted **N**) coded implicitly or explicitly in a machine instruction statement for an SS-type instruction, from which the Assembler derives a *Length Specification Byte* **L** or a *Length Specification Digit* L_n. In this text, often described by the terms N or N_n.

length modifier
A modifier specifying the exact length to be used for a constant, rather than its default length.

Length Specification Byte
The second byte (**L**) of an SS-type instruction, one less than the true length of its operand.

Length Specification Digit
A 4-bit hexadecimal digit (**L_n**) in The second byte of an SS-type instruction, one less than the true length of its operand.

library
A general term for a file or data set to save programs or data for later access.

linear subscript
For arrays of two or more dimensions, the evaluation of a subscript that treats the array as having been mapped into a one-dimensional array corresponding to the CPU's linear arrangement of bytes in memory.

linkage convention
An agreed set of rules for transferring control between a calling and a called program, passing arguments and receiving results, and preserving caller information during the execution of the called routine so it can be restored on return to the caller.

Linkage Editor
The predecessor to the z/OS Binder; its functions are included in the Binder. A Linkage Editor is used on z/VSE.

linked list
Same as *list*. Sometimes called a "single-threaded" list.

Linker
A program that converts and combines object modules and load modules into an executable format ready for quick loading into memory by the Program Loader.

linking loader
Links and places modules directly into storage with linking, immediately prior to program execution.

list
(1) A sequence of data elements each containing a link to its successor. If the first and last elements are identified and the next element to be accessed is the last (most recently added, sometimes called a "First In, First Out" (FIFO) list or *stack*. (2) Colloquially, a table.

literal
A special symbol with the side effect of defining a constant referenced by that symbol.

literal pool
A set of literal-generated constants grouped together by the Assembler. A program may contain multiple literal pools.

Little-Endian
A representation of numbers in which the value of the digits at successively higher addresses have *greater* significance; the digits have increasing significance from left to right.

load module
(1) A generic name for the output of a Linker; a mixture of machine language instructions and data ready to be saved in a *library*, or to be loaded directly into memory for execution. (2) The original form of System/360 executable, stored in a Partitioned Data Set (PDS) program library in "record format".

load operation
Replace the contents of a register with a copy of data from a memory address or from another register. Other parts of the register may contain sign-extended bits (for arithmetic loads), or zero-extended bits (for logical loads). The original contents of the target register are not preserved.

Loader
(1) On z/VM systems, a program that can link object modules directly into memory for execution, or generate a relocatable MODULE. (2) On older OS/360 systems, a program that links object and load modules into memory for execution; now called the "Batch Loader". (3) The Program Loader.

location
A position within the object code of an assembled program, as determined by assigning values of the *Location Counter* during assembly. An *assembly time* value, sometimes confused with an *execution time address*.

Location Counter (LC)
A counter used by the Assembler at assembly time to construct its model of the relative positions of all components of an assembled program.

logical arithmetic
Binary arithmetic and comparison operations with unsigned operands.

logical division
Division of two unsigned operands, generating an unsigned quotient and unsigned remainder.

logical multiplication
Multiplication of two unsigned operands, generating an unsigned product.

logical operation
An operation between individual bits; one of AND, OR and Exclusive OR (XOR).

logical representation
An unsigned number representation.

logical shift
A movement of bits in a general register to the left or right, inserting zero bits into any vacated bit positions.

M

M_n, m_n
Mask field in machine instruction operand n.

machine language
The binary instructions and data interpreted and manipulated by the processor when a program is executed.[326] Compare *Assembler Language*.

machine length
An Encoded Length, one less than the Length Expression (program length) coded in the instruction statement.

macro instruction
A powerful means to encapsulate groups of statements under a single name, and then generate them (with possible programmer-determined modifications) by using the macro-instruction name as an operation field entry. Often abbreviated "macro".

mantissa
A term previously used to describe the significand of a floating-point number. Because it can be confused with the mantissa (fractional part) of a logarithm, avoid its use when describing floating-point representations.

mask
(1) A bit in an instruction controlling its behavior. (2) A bit in the FPCR controlling the actions to be taken when an exception condition occurs. (3) The Program Mask in the PSW.

MaxReal
The largest representable floating-point magnitude, also called Max.

MBCS
Multiple-Byte Character Set, in which characters are represented by one to four bytes.

message character
Any character in an edit pattern that is not a Digit Selector (DS), a Field Separator (FS), or a Digit Selector and Significance Starter (SS).

millicode
Internal instructions used by the CPU to perform operations too complex to be done cost-effectively in hardware. (Sometimes erroneously called "microcode".)

MinReal
The smallest representable floating-point magnitude. If normalized, it is also called Min; if denormalized, it is also called DMin.

minuend
see *subtrahend*

mnemonic
A character string representing an instruction name, intended to be easier to remember than the *operation code* of the instruction. A convenient shorthand for the name of an instruction. For example, the "Branch and Save" instruction has mnemonic "BAS".

modal instruction
An instruction that places or updates addresses in the general registers, with results that depend on the *addressing mode*.

modifier
A value following the constant type, specifying other information about the constant's Length, Scale, and Exponent.

multiplicand
In a multiplication, the number that is to be multiplied (the first operand) by another, the multiplier (the second operand)

multiplier
See *multiplicand*

multiply and add/subtract
An instruction in which a double-length product is created internally to which a third operand is added or subtracted before truncating or rounding the result to the length of the original operands.

326 It is not meant to be intelligible to normal human beings.

N

N, N_n
The Length Expression you specify in an SS-type instruction, giving the true length of an operand. The Assembler converts that value to the *Length Expression* used by the CPU when executing the instruction.

N'
Number attribute reference to a conditional assembly symbolic parameter or dimensioned SET symbol (as in N'&PARAM).

NaN
A floating-point "Not-a-Number" having no numeric or mathematical meaning.

nominal value
The value you write between delimiters or value separators to specify the assembled value of a constant.

non-overflowed zero
A calculated zero result that is not the result of an overflow condition.

no-operation instruction
An executable instruction having no effect other than to occupy space, usually to align a following instruction on a desired boundary.

normalization
A process of ensuring that the most significant digit in a fraction-based floating-point representation is nonzero.

null byte
A zero or X'00' byte, sometimes indicated by the character ***n***. Used to terminate a C-string.

numeric digit
The rightmost 4 bits of a byte.

O

object code
The machine language contents of an object module.

object module
Records containing the external symbols, machine language text, and relocation dictionary information required for program linking, in either the 80-byte card image "OBJ" format or in the GOFF format.

offset
(1) The MVO instruction shifts or *offsets* the second operand to the left by one hex digit before appending it to the sign digit of the first operand. (2) A field in a relative-immediate instruction giving the distance in halfwords from the instruction. (3) Sometimes used coloquially to refer to *displacement*.

ones' complement representation
A signed binary representation where negative numbers are represented by changing each 0 bit to a 1 bit and vice versa.

opcode
An abbreviation for *operation code*. Sometimes used when the term *mnemonic* is meant.

operand
(1) Something operated on by an instruction. (2) A field in a machine instruction statement.

operand order dependence
The results of many packed decimal arithmetic instructions depend on the order of the operands. For example (007+)+(7+) yields 014+, but (7+)+(007+) causes a decimal overflow.

operation code
The z/Architecture definition of that portion of an instruction specifying the actions to be performed by the CPU when it executes the instruction. Often abbreviated "opcode", sometimes mistakenly used to refer to a "mnemonic".

operator
A character specifying a mathematical operation. One of * (meaning multiplication), / (meaning division), + (meaning addition), or − (meaning subtraction).

options
Directives to the *Assembler* specifying various "global" controls over its behavior. Options are specified by the user as a string of characters, usually part of the command or statement that invokes the assembler, or on *PROCESS statements. Some options may be dynamically modified by ACONTROL statements.

OR operation
A logical (boolean) operation between two bits, whose result is 1 if either operand bit is 1.

order dependence
Results of a packed decimal operation can depend on the order in which the operands are specified.

Ordinary USING
A USING statement directing resolution of unqualified implicit references to a specific base register.

ORG Extended Syntax
Additional operands on ORG statements that allow Location Counter alignment to a specific power-of-two boundary, and an offset from that position.

origin
A starting value assigned by you (or by the Assembler), used to calculate positions, offsets, and displacements in your program. (Because most programs are relocated, it's rarely necessary to specify an origin.)

overflow
The sum, difference, product, or quotient of two numbers is too large to be correctly represented in the number of digits or range of values available.

overflowed zero
The addition or subtraction of binary or packed decimal operands for which an overflow generates a zero result.

P

padding
Extra bits or bytes added to a constant by the Assembler so that it will fill the space allotted to it.

parameter
A place-holder in a called program, to be assigned a value from an argument provided by a calling program.

parameterization
A valuable technique for adding flexibility and generality to program definitions, typically by defining assembly-time values in EQU statements.

pattern character
Any byte in an edit pattern.

payload
Diagnostic information contained in the significand of a NaN.

pipeline
A technique used in modern CPUs to speed instruction execution by dividing the fetch. decode, and execute phases into smaller stages.

PM
See *Program Mask.*.

postfix notation
A representation of expressions convenient for evaluation. The infix form `2*(3+4)` is represented as `2 3 4 + *`.

post-normalization
A process of normalizing a floating-point operand after operating on it.

postorder tree traversal
A technique for traversing a binary tree, visiting first the left subtree, then the right subtree, and then the parent node.

precision
(1) The number of digits that can be held in a register or memory field. (2) The number of significant digits in a numeric value. Not the same as accuracy, which defines the correctness of the digits.

preferred exponent
The exponent of the result of a decimal floating-point numeric operation has a preferred value, either that of one of the operands or a value providing the maximum number of significant digits.

preferred quantum
The quantum selected for the result of a decimal floating-point operation that maximizes the number of significant digits, including low-order zero digits. Equivalent to *preferred exponent*.

preferred sign code
For packed and zoned decimal numbers, there are six valid sign codes: `X'A'`, `X'C'`, `X'E'`, and `X'F'` indicate +, and `X'B'` and `X'D'` indicate −. The preferred codes are `X'C'` and `X'D'`; these are the sign codes generated by packed decimal arithmetic operations.

preorder tree traversal
A technique for traversing a binary tree, visiting first the parent node, then the left subtree, and then the right subtree.

pre-normalization
A process of normalizing floating-point operands before operating on them.

problem state
A state in which the CPU disallows the execution of certain instructions.

program interruption
An interruption condition caused by an executing program that can be handled by the interrupted program.

program length
A Length Expression, not necessarily the length of a program. The number of bytes the programmer specifies, not the machine length encoded into the instruction.

program linking
The process of resolving external names into offsets or addresses; combining multiple input name spaces into a composite output name space.

Program Loader
The component of the Operating System that brings load modules into memory, makes final relocations, and transfers control to the program.

Program Mask
A 4-bit field in the PSW used to control whether or not certain types of exception condition should cause an interruption or take a CPU-defined default action.

program object (PO)
A newer form of executable on z/OS, stored in a PDSE (Partitioned Data Set Extended) program library.

PseudoRegister (PR), External Dummy (XD)
A PseudoRegister or external data item having length and alignment attributes. Space in the loaded module is reserved for Common control sections; space for external dummy sections must be obtained at execution time. (See *External Dummy Section*.)

PSW
Program Status Word, containing information about the current state of a program.

Q

QNaN
A Quiet Not-a-Number that does not cause an exception condition in any floating-point arithmetic operation.

Q-type address constant
A field containing the offset (not the address) of a Dummy External Symbol (or PseudoRegister) from the start of a virtual area mapped at link time and allocated at execution time.

qualified symbol
A symbol prefixed by a qualifier and separated from it by a period, as in `qualifier.symbol`.

qualifier
A symbolic identifier defined in the name field of a Labeled USING statement. It may be used only as a qualifier, and not as an ordinary symbol.

quantum
The value of a unit in the low-order digit of a decimal floating-point number.

queue
A sequence of data elements each containing links to its successor and to its predecessor. Sometimes called a “doubly-linked list.”

quotient
The primary result of a division operation.

R

R_n, r_n
(1) Register specification digit for machine instruction operand n. (2) The field of a machine instruction designating the number of a general register.

Rr_n
The GPR designated by register number R_n.

R(r1+1)
The GPR whose number is found by adding 1 to the even-numbered register specification digit r1.

R(r3|1)
The GPR whose number is found by forcing the low-order bit of the register specification digit r_3 to be a 1-bit. Thus, R(6|1) means R7, and R(7|1) means R7.

R_3|1
A notation referring to the general register containing the comparand of a branch on index instruction. If the R_3 operand is even, R_3|1 is the next higher odd-numbered register; and if the R_3 operand is odd, R_3|1 is that odd-numbered register.

radix
The base in which the significant digits of fixed-point or floating-point numbers are represented.

radix-complement representation
A signed representation where the the complement of a number is formed by complementing each digit with respect to the radix minus 1, and then adding 1 to the lowest-order digit. The numerically significant high-order digit usually contains sign information.

real address
The "true" (not *virtual*) address of a byte in memory.

"real" numbers
A powerful abstraction used by mathematicians; numbers with unlimited range and precision.

"realistic" numbers
Numbers used for computation, having finite range, precision, and accuracy.

reentrant
See *reenterable*.

reenterable
A program is reenterable if (1) Its execution can be suspended, then executed by other processes, and then resumed by the original process with correct behavior for all processes; (2) It can be executed simultaneously by multiple processes, with correct behavior for all processes. (3) Capable of simultaneous execution by two or more asynchronously executing processes or processors, with only a single instance of the program image. Typically, reenterable programs do not modify themselves, but this is neither a necessary nor sufficient condition for reenterability.

reference control section
A control section containing no machine language instructions or data, defined by a DSECT, COM, or DXD instruction.

relative address
An Effective Address determined by an offset *relative* to the location of an instruction containing a relative-immediate operand.

relocatable
A property of a program allowing it to execute correctly no matter where it is placed in memory (respecting alignment requirements) by the Program Loader.

relocate
Assign actual-storage or module-origin-relative addresses to address constants.

relocating loader
Places modules into storage *and* adjusts addresses to their correct "final" value.

relocation
A procedure used by the Linker and the Program Loader to ensure that addresses in a relocatable loaded program are correct and refer to the intended targets, no matter where it is loaded. This usually requires assigning true execution-time addresses to parts of a program.

Relocation Dictionary
A summary of each relocatable address constant in an assembly, displayed in the Assembler's listing and encoded in the RLD records of the generated object module.

remainder
The residual portion of a division left over when a dividend cannot be evenly divided by a divisor. Smaller in magnitude than the divisor.

RENT
An Assembler option requesting simple tests be made for conditions of obvious self-modification of the program being assembled. These tests are also done for control sections declared by the RSECT statement.

return address
The address of an instruction to which control should be passed when a called routine completes its execution.

return code
A small integer value (usually a multiple of 4), often placed in GR15 prior to returning to a calling program.

RMODE
Residence mode, an indication of the desired placement in memory of a Control Section or class.

rotating shift
A movement of bits in a general register to the left in such a way that bits moved out of the high-order bit position are inserted into the low-order bit position. (Also called a "circulating" shift.)

rounding digit
An extra digit used to help correctly round a calculated floating-point result.

Rounding Mode
(1) A field in the FPCR indicating the rounding action to be taken after a binary floating-point operation. (2) A field in an instruction specifying the rounding to be performed by the instruction, independent of any rounding mode specified in the FPCR.

rounding-mode suffix
A suffix with the letter R and a number, appended to the numeric value of a floating-point constant specifying the rounding to be used when converting the

nominal value to hexadecimal, binary, or decimal floating-point.

rounding modifier
A field in an instruction specifying the type of rounding to be performed on the result of a floating-point operation.

row order
A way to store arrays so that the elements of each row follow one another in memory. For arrays of two or more dimensions, subscripts cycle most rapidly from right to left.

row-major order
Same as *row order*.

RSECT
A *reenterable control section*, distinguished from an ordinary *control section* (CSECT) only by (a) the presence of a flag in the *External Symbol Dictionary* and (b) that High Level Assembler will perform *reenterability* checking of instructions within the RSECT.

run time
See *execution time*.

S

S'
Scale attribute reference to a symbol (as in S'ABC), or to a macro parameter (as in S'&PARAM).

S_n
Implied address of machine instruction operand n.

SBCS
See *Single-Byte Character Set*

scaled arithmetic
Methods for doing arithmetic with non-integer values, using fixed-point arithmetic instructions such as binary integer and packed decimal.

section
(1) A generic term for control section, dummy section, common section, etc.; a collection of items that must be bound or relocated as an indivisible unit. (2) A collection of *elements* belonging to specified *Classes* in a program object. Elements defined by a section are added or deleted as a group. (A program object section is not the same as a control section.)

segment
A component of a program object containing classes with the same properties such as RMODE and loadability.

self-defining term
One of binary, character, decimal, graphic, and hexadecimal. Its value is inherent in the term, and does not depend on the values of other items in the program.

self-modification
A program modifies its instructions or constants. Considered a very poor programming practice with severe execution-time performance penalties, and usually forbidden if the program must be reenterable.[327]

Shift-In
An X'0F' byte code in a stream of DBCS byte pairs indicating that the following single bytes are SBCS-encoded. A change from double-byte mode to single-byte mode.

Shift-Out
An X'0E' byte code in a stream of SBCS bytes indicating that the following pairs of bytes are DBCS-encoded. A change from single-byte mode to double-byte mode.

sign extension
The process of copying the sign bit of a shorter binary operand and extending it to the left, to the length of a target field.

significance exception
In hexadecimal floating-point, the result of an addition or subtraction yields a significand of all zero digits. This exception can either cause a program interruption with IC=14, or can be masked to produce a true zero.

significance indicator
An internal bit used by the CPU during ED and EDMK instructions to control subsequent editing operations.

significance starter
An edit-pattern character that sets the Significance Indicator ON. Also known as a “digit selector and significance starter” because it selects a digit if the Significance Indicator is already on or if the digit is nonzero.

significand
The numerically significant digits of a floating-point number, whether explicitly or implicitly represented.

sign-magnitude representation
The familiar signed representation of numbers with prefixed or suffixed + or − signs attached to the number's magnitude.

simple relocatability
A property of an assembly-time symbol or expression whose value changes by the same amount as a change to the program's assumed origin.

Single-Byte Character Set
A character set in which characters are represented by a single byte.

SNaN
A Signaling Not-a-Number that causes an exception condition in an arithmetic operation.

space
A nonempty, finite-width invisible character; a blank. In contexts where explicit spaces appear, we sometimes use the “•” character.

special value
A floating-point zero, a denormalized number, an infinity, a QNaN, or an SNaN.

327 Technically, a self-modifying program can be reenterable if every execution instance makes exactly the same modifications. This is considered an even poorer practice.

stack
A data structure with a single visible element, the "stack top". Sometimes called a "Last In, First Out" (LIFO) list or queue.

statement
The contents of the records read and processed by the Assembler. There are four types: comment statements, machine instruction statements, assembler instruction statements, and macro-instruction statements.

statement field
One of the four fields of an Assembler Language statement (other than a comment statement): the name, operation, operand, and remarks fields. Which fields are required and/or optional depends on the specific statement.

status flag
A bit in the FPCR indicating that an exception condition has occurred.

store operation
Place a copy of part or all of a register's contents into memory.

subtrahend
When one number is subtracted from another, the number being di**min**ished (the first operand) is the **min**uend, and the number being **sub**tracted (the second operand) is the **sub**trahend.

supervisor state
A state in which the CPU allows the execution of all instructions.

symbol
A name known at assembly time, to which various values are assigned. The values may be absolute, simply, or complexly relocatable, or determined at link or execution time.

symbol attribute
Information about the properties of a symbol, including value, relocation, length, type, scale, and integer. (Only the first three attributes are important for most uses.)

Symbol Attribute Reference
(1) A term whose value is that of a symbol's attribute. The types of Symbol Attribute Reference are length, scale, integer, definition, type, and opcode. (2) Conditional assembly attribute references include the above six, plus count, number, assembler type, and program type.

Symbol Table
A table used by the Assembler to hold the names, values, and attributes of all symbols in a program.

Syntactic Character Set
A set of 82 characters with the same encodings across all EBCDIC Code Pages.

T

T'
Type Attribute Reference to a conditional assembly symbolic parameter (as in T'&PARAM).

table
A term often used to describe a one-dimensional array whose columns may contain a mixture of different data types and lengths. (See *array*.)

target instruction
(1) The instruction to which a branch instruction might transfer control. (2) An instruction addressed by an Execute instruction.

term
A symbol, self-defining term, Location Counter reference, literal, or symbol attribute reference.

text
(1) The instructions and data generated by an assembly, encoded in the TXT records of an object module. (2) The portions of an object module containing machine language instructions and data. (3) A program object class attribute indicating that locations within the class may contain and/or be the target of address constants.

true addition
The addition of binary or packed decimal operands of like sign.

true decimal addition
The addition of packed decimal operands of like sign.

truncation
(1) Removing bits or bytes from a constant so that it will fit in the space allotted to it. (2) A method of "rounding" numeric results or constants by discarding digits or characters beyond the length of the representation.

two's complement
A method for negating a binary integer, by converting each 1 to 0 and each 0 to 1, and then adding a low-order 1.

two's complement representation
A signed binary representation where the high-order bit contains sign information, and has weight -2^{n-1}.

type extension
A second letter following the constant type, providing additional information about the constant's length or representation.

U

UCS
Universal Character Set. UCS-2 was a predecessor of Unicode that defined only 65,536 values. Sometimes used as an alternative name for UTF.

ulp
An abbreviation for "unit in the last place", a measure of the relative precision of a floating-point number.

unbiased rounding
A technique for avoiding the inaccuracies introduced by biased rounding, typically by rounding results exactly half way between two representable results to the value with an even low-order digit.

Unicode
An international standard encoding of (almost) all characters, represented as groups of 8-bit bytes (UTF-8), one or two byte pairs (UTF-16), or 32-bit units (UTF-32).

Unicode numeric characters
Unicode characters with representations between X'0030' and X'0039'.

unnormalized add/subtract
Hexadecimal floating-point addition and subtraction in which the result is not normalized.

unnormalized number
A hexadecimal floating-point number in which the significand is unnormalized.

USING statement
A promise to the Assembler that base-displacement addressing fields can be derived correctly from the base location and base address information provided in the statement.

USING Table
An internal table used by the Assembler to hold information provided in USING instructions.

UTF
Unicode Transformation Format. Three encodings (UTF-8, UTF-16, and UTF-32) that are easily converted ("transformed") among one another.

V

V-type address constant
A field containing the address of an external symbol, resolved during linking and loading.

virtual address
The apparent address of a memory location that may physically reside at a different real address.

virtual origin
The address of a (possibly nonexistent) array element all of whose subscripts are zero.

X

X_n, x_n
Index register specification digit for machine instruction operand n.

XOR operation
A logical (boolean) exclusive-OR operation between two bits, whose result is 1 if either operand bit is 1 while the other is zero. If the operand bits are identical, the result is zero.

Z

zero duplication factor
A duplication factor that causes Location Counter alignment without generating a constant. Skipped bytes are zeroed for DC instructions if the immediately preceding byte contains object code.

zero extension
The process of adding zero bits to the left of a shorter operand, to extend it to the length of a target field.

zone digit
The leftmost 4 bits of a byte.

zoned digit
An unpacked or edited packed decimal numeric digit.

Bibliography

```
BBBBBBBBBBB   IIIIIIIIII  BBBBBBBBBBB
BBBBBBBBBBBB  IIIIIIIIII  BBBBBBBBBBBB
BB        BB      II      BB        BB
BB        BB      II      BB        BB
BB       BB       II      BB       BB
BBBBBBBBBB        II      BBBBBBBBBB
BBBBBBBBBB        II      BBBBBBBBBB
BB       BB       II      BB       BB
BB        BB      II      BB        BB
BB        BB      II      BB        BB
BBBBBBBBBBBB  IIIIIIIIII  BBBBBBBBBBBB
BBBBBBBBBBB   IIIIIIIIII  BBBBBBBBBBB
```

Basic References

These are useful references you may want to have available. They should be available at these web sites:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

`http://www.ibm.com/servers/eserver/zseries/zos/webqs.html`

`http://www-03.ibm.com/software/products/en/hlasm`

- z/Architecture Processor
  - *z/Architecture Principles of Operation*, SA22-7832
  - z/Architecture Reference Summary, SA22-7871
- High Level Assembler
  - High Level Assembler Language Reference, SC26-4940
  - High Level Assembler Programmer's Guide, SC29-4941
- z/OS System Services
  - z/OS MVS Programming: Assembler Services Guide, SA23-1368
  - z/OS MVS Programming: Assembler Services Reference, Volumes 1-2, SA23-1369 – SA23-1370
  - z/OS MVS Programming: Extended Addressability Guide, SA22-7614
  - z/OS MVS System Codes, SA22-7626
  - z/OS MVS System Messages, Volumes 1-10, SA22-7631 – SA22-7640
- z/OS Data Areas and Control Blocks
  - z/OS MVS Data Areas, Volumes 1-6. GA32-0853 – GA32-0858
- z/OS Input/Output
  - z/OS DFSMS Using Data Sets, SC26-7410
  - z/OS DFSMS Macro Instructions for Data Sets, SC23-6852
- z/OS Binder
  - z/OS MVS Program Management User's Guide and Reference SA22-7643
  - z/OS MVS Program Management Advanced Facilities SA22-7644
- z/VM
  - z/VM CMS Application Development Guide for Assembler, SC24-6070

- z/VSE
  - z/VSE Guide to System Functions, SC33-8312
  - z/VSE Messages and Codes, Volumes 1-3, SC33-8306 – SC33-8308

System/360 Architecture History

- IBM System/360 Principles of Operation, Form A22-6821.
- Architecture of the IBM System/360, by G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. IBM Journal of Research and Development, Volume 8, No. 2, April 1964. (Reprinted in IBM Journal of Research and Development Vol. 44 No. 1/2, January/March 2000.)
- The Structure of System/360, a series of five articles in the IBM Systems Journal, Volume 3, Number 2, 1964.
  - Part I - Outline of the Logical Structure, by G. A. Blaauw and F. P. Brooks, Jr.
  - Part II - System Implementations, by W. Y. Stevens.
  - Part III - Processing Unit Design Considerations, by G. M. Amdahl.
  - Part IV - Channel Design Considerations, by A. Padegs.
  - Part V - Multisystem Organization, by G. A. Blaauw.
- D. W. Sweeney, *An Analysis of Floating-Point Addition*, IBM Systems Journal Vol. 4 No. 1 (1965)
- The functional structure of OS/360 Part I: Introductory Survey. IBM Systems Journal Vol. 5 No. 1, 3-11 (1966)
- The functional structure of OS/360 Part II: Job and Task Management. IBM Systems Journal Vol. 5 No. 1, 12-29 (1966)
- The functional structure of OS/360 Part III: Data Management. IBM Systems Journal Vol. 5 No. 1, 30-51 (1966)
- Microprogram Control for System/360, by S. G. Tucker. IBM Systems Journal, Volume 6, Number 4, 1967.
- Lisa Heller and Mark Farrell, *Millicode in an IBM zSeries processor*, IBM Journal of Research and Development, Volume 48, Number 3/4, May/July 2004.

Assembler Design and Implementation

- IBM System/360 Operating System Assembler Language, Form C28-6514.
- Macro Language Design for System/360, by D. N. Freeman. IBM Systems Journal, Volume 5, Number 2, 1966.
- Proceedings of the IBM Macro Assembler Conference, May 1-3, 1967, Los Gatos, California. (This collection of nine papers is an interesting source of “inside” information presented by assembler specialists in the computer industry.)
- PL/360, A Programming Language for the 360 Computers, by Niklaus Wirth. Journal of the ACM, Volume 15, January 1968.
- Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM 360, by William Kent. Computing Surveys, Volume 1, Number 4, December 1969.
- Assembler Design and its Effect on Language and Performance, by H. Joseph Myers. Proceedings of SHARE XXXIV, March 1970, Denver, Colorado.
- A Brief System z Assembler History, by John Ehrman. Proceedings of SHARE 120, Session 12235, San Francisco, California, February 2013.

Other General References

- Donald Knuth, *The Art of Computer Programming*, Addison-Wesley.
- Pat Sterbenz, *Floating-Point Computation*, Prentice-Hall, 1974.

Acknowledgments

```
 AAAAAAAAAA    CCCCCCCCCC   KK        KK
AAAAAAAAAAAA  CCCCCCCCCCCC  KK       KK
AA        AA  CC        CC  KK      KK
AA        AA  CC            KK     KK
AA        AA  CC            KK    KK
AAAAAAAAAAAA  CC            KKKKKKK
AAAAAAAAAAAA  CC            KKKKKKK
AA        AA  CC            KK    KK
AA        AA  CC            KK     KK
AA        AA  CC        CC  KK      KK
AA        AA  CCCCCCCCCCCC  KK       KK
AA        AA   CCCCCCCCCC   KK        KK
```


My thanks to the following:

ASSEMBLER_LIST discussion group for many interesting suggestions and observations.

Avri Adleman for many useful examples of advanced programming techniques, and for continued encouragement.

William Blair for reviewing several chapters.

Bryan Childs for several helpful suggestions.

Richard Corak for thorough and helpful comments on many drafts.

John Dravnieks for helping me understand the High Level Assembler, and for being a valued colleague for many years.

John Ganci for permission to use several clever solutions to programming exercises, and for his thorough, sharp-eyed and knowledgable reviews of the entire text, and many thoughtful suggestions.

Dan Greiner for many tutorials on the behavior of System z and its instructions, and for thorough and detailed reviews of drafts.

Vicki Griffeth for helping me to see an early version of these notes through a reader's eyes.

David Gross for thorough proofreading and many helpful suggestions.

John Kalinich for reviewing several chapters.

Melvyn Maltz for suggestions and very careful proof-reading.

Lisa Moore for helping me understand and clarify the description of character encodings, especially Unicode.

Ed Oddo, William Deason, Ken Edwards, and Virgil Hein for supporting my efforts to complete these notes.

Prof. Joshua Panar of Ryerson University for extensive thoughtful comments, and “field testing” in a university environment.

Wayne Rhoten for help clarifying many important points.

Michael Stack of Northern Illinois University for his inspiration as a teacher of Assembler Language and as the creator of the popular SHARE “Assembler Boot Camp”.

Rich Way for catching a variety of my errors.

Romney White for thorough and helpful comments on many drafts.

Stanford University students for their critiques of a very early version of these notes.

Notices

```
NN         NN  000000000000  TTTTTTTTTTTT
NNN        NN  000000000000  TTTTTTTTTTTT
NNNN       NN  00        00       TT
NN NN      NN  00        00       TT
NN  NN     NN  00        00       TT
NN   NN    NN  00        00       TT
NN    NN   NN  00        00       TT
NN     NN NN  00        00       TT
NN      NNNN  00        00       TT
NN       NNN  00        00       TT
NN         NN  000000000000       TT
NN          N  000000000000       TT
```

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to those patents. You can send license inquiries, in writing, to

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law. INTERNATIONAL BUSINESS MACHINES PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

BookMaster
Enterprise Systems Architecture/390
ESA/390
MVS
OS/390
S/370
System/370
System z
z/Architecture
z/VM
zLinux
z System
Enterprise Systems Architecture/370
ESA/370
IBM
OS/360
S/360
System/360
System/390
VSE/ESA
z/OS
z/VSE
zSeries

ANSI is a registered trademark of the American National Standards Institute in the United States, other countries, or both.

IEEE is a trademark of the Institute of Electrical and Electronic Engineers in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Unicode is a registered trademark of Unicode, Incorporated in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company product and service names may be trademarks or service marks of others.

Suggested Solutions to Selected Exercises and Programming Problems

Section 1 Solutions

Section 1.2

1.2.1. The field is 8 digits wide.

Section 1.3

1.3.1. The reference is to GR9.

Section 2 Solutions

Section 2.1

2.1.1. (a) 22, (b) 44, (c) 170, (d) 127.

2.1.2. 2^n.

2.1.3. Assuming the number is unsigned, $2^n - 1$. Later, when we discuss signed numbers in the twos' complement representation, the answer will be -1.

Section 2.2

2.2.1. (a) `B'1010'`, (b) `B'10 1101'`, (c) `B'11 1110 1000'`.

2.2.2.

Binary Digits	Decimal Value	Hex Digit	Octal Digits
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	10	A	12
1011	11	B	13
1100	12	C	14
1101	13	D	15
1110	14	E	16
1111	15	F	17
10001	16	10	20

2.2.3. Groupings always begin at the radix point, which for integers lies to the right of the last digit. (Consider what would happen to `B'111111'` if you grouped the bits from the left: should the result be `B'1111 1100' = X'FC'`, rather than `X'3F'`?)

2.2.4. These are the hexadecimal addition and multiplication tables.

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

×	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	00	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	00	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	00	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	00	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	00	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	00	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	00	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	00	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	00	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	00	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	00	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	00	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

2.2.5.

1. 21474 (base 8) = X'233C'.
2. 77777 (base 8) = $2^{15}-1$ = X'7FFF'.
3. 1750 (base 8) = X'3E8'.
4. 60341303 (base 8) = X'C1C2C3'.
5. 4631 (base 8) = X'999'.

Section 2.3

2.3.1.

1. 26293 = X'66B5' = B'110 0110 1011 0101' = 63265 (base 8) = 12122311 (base 4).
2. X'2FACED' = B'10 1111 1010 1100 1110 1101' = 3124461.
3. X'BABEF00D' = 27257570015 (base 8) = 3133075469.
4. X'C0FFEE' = B'1100 0000 1111 1111 1110 1110' = 12648430.

2.3.2.

1. 2147483647 (a useful number!)
2. 12698307
3. 1077952604

2.3.4.

1. X'257' = 599
2. X'7FFA' = 32762
3. X'8008' = 32776
4. X'E000' = 57344
5. X'FFFA' = 65530
6. X'E1010' = 921616

2.3.5. It is usually most convenient to do the arithmetic in base A (the original base) by expressing B in base A. Sometimes it is simpler to convert from base A to base 10, and from there to B using decimal arithmetic. (We can't do the arithmetic in base B, since we would have to know the representation of the number to be converted *in base B* in order to do the arithmetic!) The result does not depend on the base used for the conversion.

2.3.6. (a) 5061 (octal) = 2609 (decimal); (b) 111, (c) 192 is not a valid octal number (9 is not an octal digit!).

2.3.7. Referring to the tables of powers of 16, we find 9K = 589,284; 5M = 83,866,080; 2G = 2,147,483,648.

Section 2.4

2.4.1.

1. 31659 = 75653 (base 8) = 13232223 (base 4) = B'111 1011 1010 1011'.
2. 6917 = 210132 (base 5) = 31C1 (base 13) = X'1B05'.
3. X'EF2A' = 61226 = 21B39 (base 13).

2.4.2. The hex values are 1, A, 64, 3E8, 2710, 186A0, F4240, 989680, 5F5E100, 3B9ACA00. Did you do many tedious base-10 divisions by 16, rather than nine base-16 multiplications by X'A'? Try it, it's good practice in hex arithmetic. (In this way, you can derive further values easily; the next three are 2540BE400, 174876E800, and E804A51000.)

2.4.4. Your answer should be 1000 (decimal).

2.4.5. 73294 = 58001 (base 11) = 374BA (base 12) = 27490 (base 13) = 1C9D4 (base 14) = 16AB4 (base 15). The previously computed results are of little use in converting to each new base.

2.4.6. Here is a base-7 multiplication table:

×	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	11	13	15
3	3	6	12	15	21	24
4	4	11	15	22	26	33
5	5	13	21	26	34	42
6	6	15	24	33	42	51

(1) 526 (base 7) = 265 = X'109'. (2) 11010 (base 7) = 2751 = 5277 (base 8). (3) 61436 (base 7) = 35174 (base 8). (4) 666 (base 7) = 342, since 666 in base 7 is the same as 7^3-1, or $343-1$.

2.4.7. 757 (base 10), 531 (base 12).

2.4.8. (1) 2F3 (base 25) = 1628. (2) 61436 (base 8) = 25374. (3) X'DEFACE' = 14613198. (4) 999 = X'3E7'.

2.4.9. Whatever base is convenient; base A is most natural; occasionally it is simpler to use decimal arithmetic.

2.4.10. (1) 526 = X'10E'. (2) B'10110' = 42 (base 5). (3) 61436 (base 8) = 25374. (4) 666 = 1641 (base 7).

2.4.11. 10, 11, 12, 13, 20, 22, 1000, 00000000 (or eight marks of some sort; the only reason to write zeros is because the numeric "digit" available in base 1 is zero).

2.4.12. One base must be a power of the other. (This doesn't necessarily mean it will be easy!)

2.4.13. The values are 11, 12, 13, 14, 15, 16, 17, 20, 22, 24, 31, 100, and 121. (You may have seen this sequence as a puzzle question "What is the next number in this series?")

2.4.14. $435_7 = 222_{10}$, and $64_7 = 46_{10}$. Their sum in base 7 is 532, or 268 in decimal. Their base-7 product is 41526, or 10212 in decimal.

2.4.15. The base-3* values are respectively 2, 20, 101, 121, 1001, 1112, 10200, and 10212.

Section 2.6

2.6.1.

1. X'DEADBEEF' = 3735928559
2. X'FFFFFFFF' = 4294967295
3. X'DECODED1' = 3737181905

Section 2.7

2.7.1. The quotients and remainders for each division step are:

1. X'B675', X'3';
2. X'C29', X'E';
3. X'CF', X'8';
4. X'D', X'C';
5. X'0', X'D'.

The converted value of X'ABODE' is DC8E3 (base 15).

2.7.2. (1) 0729, (2) 9271, (3) 9999, (4) 9999 (not valid), (5) 5000, (6) 5000 (not valid), (7) 0001 (not valid). Valid values in the 4-digit ten's-complement representation must lie between −5000 (the maximum negative number) and +4999 (the maximum positive number).

2.7.3. $-2^{11}+1$, or -2047.

2.7.4. $-2^{(n-1)}+1$.

* Mathematician's joke: There are three types of people: those who can count, and those who can't.

2.7.5. The three values are

1. X'DEADBEEF' = −559038737
2. X'FFFFFFFF' = −1
3. X'DECODED1' = −557785391

Section 2.8

2.8.1. Because it was chosen that way. It would be awkward to use a representation in which the process of converting a positive number to negative was different from the process for converting from negative to positive.

2.8.2. (See Exercise 2.4.4 also)

1. X'0257' = 599
2. X'7FFA' = 32762
3. X'8008' = –32760
4. X'E000' = –8192
5. X'FFFA' = –6

2.8.3. The two's complement of the binary representation of a number X is the two's complement binary representation of the number −X, unless X is the negative number of greatest representable magnitude.

2.8.4. In decimal, c(A) = +32064, c(B) = −12288, c(C) = +5538, c(D) = −32758.

2.8.5. The variables, their decimal values, and their 9-bit representations for both positive and negative values, are shown below. N.R. means the value cannot be represented.

		Positive	Negative
Z	0	B'000000000'	N.R.
A	1	B'000000001'	B'111111111'
B	9	B'000001001'	B'111110111'
C	62	B'000111110'	B'111000010'
D	101	B'001100101'	B'110011011'
E	255	B'011111111'	B'100000001'
F	256	N.R.	B'100000000'

2.8.6.

1. +10 X'0000000A', –10 = X'FFFFFFF6'
2. +729 = X'000002D9', –729 = X'FFFFFD27'
3. $+10^6$ = 1000000 = X'000F4240', −1000000 = X'FFF0BDC0'
4. $+10^9$ = 1000000000 = X'3B9BCA00', −1000000000 = X'C4643600'
5. +2147483648 = 2^{31} is not representable, -2^{31} = X'80000000'
6. +65535 = $2^{16}-1$ = X'0000FFFF', −65535 = X'FFFF0001'
7. +2147483647 = $2^{31}-1$ = X'7FFFFFFF', $-(2^{31}-1)$ = X'80000001'

2.8.7. The two processes give the same bit patterns. The only case where you might consider them different is in complementing the maximum negative number: our recipe would indicate an overflow in the second step (when we add a low-order 1 bit), whereas the suggested method would indicate an overflow in the first (subtraction) step. If subtraction plus complementation is thought of as a single operation, then the two procedures are the same.

2.8.8.

1. +13055 = X'32FF'
2. −9582 = X'DA92' (two's complement of X'256E')

2.8.9.

1. +5 = X'00000005'
2. −97 = X'FFFFFF9F'
3. +65795 = X'00010103'
4. −16777158 = X'FF00003A'
5. +16777219 = X'01000003'
6. −78606 = X'FFFECCF2'

2.8.10. The values are:

1. X'B00F' = -20465
2. X'FFF1' = -15
3. X'0FFF' = +4095
4. X'F001' = -4095

Section 2.9

2.9.1.

1. X'00000257' = B'0000 0000 0000 0000 0000 0010 0101 0111'
2. X'00007FFF' = B'0000 0000 0000 0000 0111 1111 1111 1111'
3. X'FFFF8008' = B'1111 1111 1111 1111 1000 0000 0000 1000'
4. X'FFFFE000' = B'1111 1111 1111 1111 1110 0000 0000 0000'
5. X'FFFFFFFA' = B'1111 1111 1111 1111 1111 1111 1111 1010'

Section 2.10

2.10.1. No. Overflow must be detected using binary arithmetic. Since the four high-order bits are B'1111', their sum is B'1 1110', so the carries into and out of the high-order *bit* positions agree.

Section 2.11

2.11.1.

1. 10−(−10) = X'0000000A' − X'FFFFFFF6' = X'00000014' (no overflow, no carry);
2. 729−65535 = X'000002D9' − X'0000FFFF' = X'FFFEFD28' (no overflow, carry);
3. 2147483647+2 = X'7FFFFFFF' + X'00000002' = X'80000001' (overflow, no carry);
4. $10^9+(-(2^{31}-1))$ = X'3B9BCA00' + X'80000001' = X'BB9BCA01' (no overflow, no carry);
5. 0−(+0) = X'00000000' − X'00000000' = X'00000000' (no overflow, carry);
6. (−10)+10 = X'FFFFFFF6' + X'0000000A' = X'00000000' (no overflow, carry).

2.11.2. C(X) = X'679E', no overflow; C(Y) = X'500A', overflow; C(Z) = X'FD4A', no overflow.

2.11.3. The procedure is correct. Consider these examples, using 4-bit signed binary numbers.

Number	Step 1	Step 2			
0000	1111	1111	0	–1	
0001	0000	0000	1	0	
0111	0111	0110	7	6	
1111	1111	1110	–1	–2	
1000	1111	0111	–8	+7	(with overflow)

2.11.5.

1. X'7D26F071'+X'B40E99A4' = X'31358A15',   carry, no overflow
2. X'7D26F071'-X'B40E99A4' = X'C91856CD', no carry,   overflow
3. X'FFFFF39A'+X'FFFE4B06' = X'FFFE3EA0',   carry, no overflow
4. X'FFFFF39A'-X'FFFE4B06' = X'0001A894',   carry, no overflow
5. X'80000003'+X'0000007C' = X'8000007F', no carry, no overflow
6. X'80000003'+X'8000007C' = X'0000007F',   carry,   overflow

Section 2.12

2.12.1. In a 9−bit hexadecimal representation,

(1) A+C = X'001+X'03E' = X'03F' (no carry, no overflow);
(2) D−E = X'065'–X'0FF' = X'166' (no carry, no overflow);
(3) Z+(−F) = X'000'+X'100' = X'100' (no carry, no overflow);
(4) (−E)−C = X'101'–X'03E' = X'0C3' (carry, overflow);
(5) (−B)+A = X'1F7'+X'001' = X'1F8' (no carry, no overflow);
(6) C−Z = X'03E'–X'000' = X'03E' (carry, no overflow);
(7) A+(−A) = X'001'+X'1FF' = X'000' (carry, no overflow).

Section 2.13

2.13.1. Consider subtracting X'80000000' from zero. If we assume that the two's complement of X'80000000' is added to the first operand (zero), then *no* carries occur out of the two high-order bit positions, and no overflow condition is detected. If we add the ones' complement of the second operand and a low-order 1-bit to the first operand, the two carries will differ and the overflow will be correctly detected. Any overflow that occurs in forming the two's comple-

ment of the second operand will be lost prior to adding if we believe the incorrect description. Similarly, suppose we subtract zero from anything. If we add the two's complement of zero, no carry occurs, but if we add its ones' complement and a low-order 1-bit, a carry always occurs. Thus “our” rule describes both the carry and overflow cases correctly.*

Section 2.14

2.14.1. The decimal values, and their corresponding logical and arithmetic representations (written as 11-bit hexadecimal numbers), are as follows; “N.R.” means that no valid representation exists to the given accuracy.

```
                Positive   Negative
  (1)    200     X'0C8'     X'0C8'
  (2)   1023     X'3FF'     X'3FF'
  (3)  -1000      N.R.      X'418'
  (4)   2047     X'7FF'      N.R.
  (5)     -1      N.R.      X'7FF'
  (6)  -1024      N.R.      X'400'
  (7)  -1023      N.R.      X'401'
  (8)   1024     X'400'      N.R.
  (9)     -0      N.R.       N.R.
```

2.14.2. The following table shows the results, where the abbreviations “C” means “Carry”, “NC” means “No Carry”, “O” means “Overflow”, and “NO” means “No Overflow”.

+	A	B	C	D
A	11110 C, NO	00001 C, NO	01111 C, O	01110 C, NO
B	—	00100 NC, NO	10010 NC, NO	10001 NC, O
C	—	—	00000 C, O	11111 NC, NO
D	—	—	—	11110 NC, O

2.14.3. Using the same abbreviations as in the previous exercise, the results will look like this (the values in the first row are A−A, B−A, C−A, D−A):

	A	B	C	D
−A	00000 C, NO	00011 NC, NO	10001 NC, NO	10000 NC, O
−B	11101 C, NO	00000 C, NO	01110 C, O	01101 C, NO
−C	01111 C, NO	10010 NC, O	00000 C, NO	11111 NC, O
−D	10000 C, NO	10011 NC, NO	00001 C, O	00000 C, NO

Section 2.15

2.15.1. (1) 0028, (2) 9951, (3) 0527, (4) 9667, (5) 8766, (6) 2469.

2.15.2.

```
(a) 0028             (b) 0527             (c) 8766
   +9951                +9667                +2469
    9979 = −21           0194                 1235
```

These calculations were simpler because no intermediate complementations were required.

2.15.3. (a) 0028, (b) 9950, (c) 0527, (d) 9666, (e) 8765, (f) 2469.

* The error wasn't corrected until the 8th edition!

Section 3 Solutions

Section 3.1

3.1.1. Because X'30A6'−X'2EC9' = X'1DD', the area contains X'1DE' or 478 bytes. Because you must consider boundary alignments for half-, full-, and doublewords, some bytes on either end can't be used. Thus, the area can contain 238 halfwords, 118 words, and 58 doublewords. (Try the same exercise with the ending address at X'30A7', and note the differences.)

3.1.2. No. The smallest addressable entity in memory is a byte. The bits within a byte are not individually addressable.

3.1.3. The area contains X'17' = 23 complete bytes, plus 3 bits at X'1A023' plus 2 bits at X'1A03B', giving 8*23+3+2 = 189 bits in all.

3.1.4. (1) halfword; (2) none; (3) halfword, word, and doubleword; (4) halfword and word.

3.1.5. (1) 131,072; (2) 32,768; (3) 2,097,152.

3.1.6. 322 in octal, and D2 in hex.

3.1.7. Let “x” mean either a 0-bit or a 1-bit. Then these bit patterns of the rightmost hex digit mean the indicated alignments:

```
0000   quadword
x000   doubleword
xx00   word
xxx0   halfword
xxx1   byte
```

Section 3.3

3.3.1. No (GR7 and GR8 are not part of an *even-odd* pair); no (for the same reason); yes.

3.3.2. Sixteen. Actually, depending on the instructions used to load data from memory into the general registers, the number can vary from 1 to 16. (Prior to System z, each general register contained only four bytes, so a pair contained eight.)

Section 3.4

3.4.1. Only one: the right half of the register is ignored for 32-bit operands.

3.4.2. Probably so that the Floating-Point Feature could be omitted if the customer didn't want it.* In many programs, the amount of arithmetic involving exchanges of data between the general and Floating-Point registers is small, so that little would be gained by using a single register set for both types of operands. For other more complex programs, however, this capability is extremely valuable.

One useful aspect of such a shared-register arrangement is that the programmer can decide the most efficient use of the registers by his program; he can allocate as many or as few to floating-point operands as he feels are needed.

* Or, as one cynic put it, so IBM could charge more for a processor with a full instruction set.

Section 4 Solutions

Section 4.1

4.1.1. You can do this if each instruction contains the address of its successor. This technique was used on the IBM 650, but it also meant that instructions had to be longer, to hold both the operand memory address and the successor instruction address.

Section 4.2

4.2.1. No, only on a halfword boundary. It is possible to require a word alignment in memory for the *operand* referred to by the instruction, but not for the instruction itself.

4.2.2. Two bytes.

4.2.3. Because (1) each instruction is an integral number of halfwords in length, and (2) can start on *any* halfword boundary.

4.2.4. Yes. A word or doubleword boundary is also a halfword boundary.

4.2.5. There's no way to tell the difference.

Section 4.3

4.3.1. No, because the first two bits of the operation code uniquely determine the length of the instruction.

4.3.2. Twenty-four (decimal) bytes. The instructions are 2, 4, 4, 4, 6, and 4 bytes long. (Types are RR, RX, RS, RX, SS, RX.)

4.3.3. In binary, the values are 01, 10, 10, 10, 11, 10.

4.3.4. 2×((sum of first two opcode bits)+1) bytes.

4.3.5. The table entries could look something like this:

```
00   RR     2   01   0
01   RX     4   10   1
10   RS,SI  4   10   1
11   SS     6   11   2
```

(But you could also study Table 10!)

4.3.6. There are seven instructions, of lengths 4, 2, 4, 6, 2, 4, and 2 bytes respectively. The types are RS, RR, RX, SS, RR, RX, and RR.

4.3.7. In most cases, you can find the required address by subtracting twice the ILC (the number of halfwords in the previous instruction) from the IA in the PSW.

4.3.8. The System z CPU will fetch whatever bytes are in the gaps, and try to decode them as instructions. There is no escape.

(You can cheat by using a branch instruction to jump over each gap, but then your program would be much larger than necessary. Some early computers handled this problem by having each instruction contain the address of its successor, but those programs were also bigger. See Exercise 4.1.1 above.)

Section 4.4

4.4.1. (1) the RR-type instruction has opcode `X'05'`. (2) the RX-type instruction has opcode `X'58'`. (3) the RS-type instruction has opcode `X'89'`. (4) the RX-type instruction has opcode `X'5A'`. (5) the SS-type instruction has opcode `X'D2'`. (6) the RX-type instruction has opcode `X'50'`.

Section 4.6

4.6.1. The Instruction Address in the PSW is odd, which violates the requirement that all instructions begin at a halfword boundary.

4.6.2. Because the New PSW's Instruction Address is odd, the attempt to fetch the first instruction at memory location `X'A237'` would cause another program interruption. The current PSW stored at the “old PSW” area would be the one shown, overlaying the “old PSW” that described the original error. The CPU would then stay in a program interruption loop until the processor was reset (which other executing programs would consider very unfriendly).

4.6.3. The causes of the Interruption Codes are:

1. 0001: an invalid instruction code that can't be executed by the CPU.
2. 0009: your program tried to divide two binary numbers; either the divisor is zero, or the quotient is too large to be represented in a single general register.
3. 000C: the product of two hexadecimal floating-point numbers is too large to be correctly represented.

Section 5 Solutions

(All answers are in hexadecimal unless otherwise indicated.)

Section 5.1

5.1.1. Yes, and don't forget it! (You can of course put an addressing halfword somewhere not on a halfword boundary, but then it can't be part of an executable instruction.)

5.1.2. Sixteen and fifteen.

Section 5.2

5.2.1. (1) X'1AAF10', (2) X'02AF0C', (3) X'000FB0'.

5.2.2. (1) X'02B00F', (2) X'000FC8', (3) X'1ABD48'.

Section 5.3

5.3.2. If the instruction type is RX, then the fourth digit of the instruction is an index digit; if it is not zero, an indexing cycle is needed.

5.3.3. For RR instructions, none; for RX, two (the base and index registers); for RS and SI, one; for SS, two (one for each operand).

5.3.4. If the instruction is type RR, or if the CPU needs repair.

Section 5.4

5.4.1. Register zero is *never* used as a base or index register. In all other respects, however, GR0 is a perfectly normal and well-behaved general register. (Later, we'll see instructions that use GR0 to hold addresses.)

5.4.2. (1) X'1AAF10', (2) X'02AF0C', (3) X'000323', (4) X'8AADD2', (5) X'000166', (6) X'000FB0'. It is important to distinguish the RX and the RS-SI instructions.

5.4.3. (1) X'0710FC' (the carry is lost!), (2) X'0903AA', (3) X'000000', (4) X'00006C', (5) X'000044', (6) X'090518', (7) X'071F49'.

5.4.4. X'FF0503' (RX instruction with base=0 and index=4, and for this particular instruction the operand could be misaligned); (2) X'31C000'; (3) X'2B4FFE' (RX instruction with base=7, index=15).

Section 5.5

5.5.1. (1) X'1BAD', (2) X'3A55' or X'0A4D', (3) not addressable, (4) not addressable, (5) not addressable.

5.5.2. It is most important to remember that *addressability depends on the contents of the general registers*. The fact that a byte has an address does not mean it is addressable; it is possible that it is inaccessible to the program. (1) not addressable (R0 can't be used as a base register); (2) X'F000'; (3) not addressable; (4) not addressable (R1 would require a displacement of 1002, and GR11 would require −1); (5) X'C3FC'; (6) X'5FC3'; (7) not addressable; (8) X'8921' *or* X'BFE6' (the Assembler has a problem deciding, too); (9) not addressable; (10) X'5000'.

5.5.3. (See the note in the solutions to Exercise 5.5.2 also.) We will write the index digit and addressing halfword as five hex digits in the form xbddd. (1) 010A20 is not addressable; (2) FFFFFF is addressable by 0F000, F0000, FF001, EF000, FE000; (3) 6A0054 is not addressable; (4) 31AB7E is addressable by FB000, BF000, 19FF2, 91FF2, 1C1CA, C11CA; (5) 001234 is addressable by 0C3FC, C03FC, FC3FD, CF3FD, 9C3EC, C93EC, EC3FC, CE3FC; (6) 07D3C4 is addressable by 05F03, 50F03, E5F03, 5EF03, F5F04, 5FF04, 5C0CC, C50CC, 95EF3, 59EF3; (7) 00A004 is addressable by DD004; (8) 31BB65 is addressable by 08921, 80921, E8921, 8E921, 98911, 89911, 0BFE6, B0FE6, BEFE6, EBFE6, 9BFD6, B9FD6, F8920, 8F920, FBFE7, BFFE7; (9) 9ABCDE is not addressable; (10) 07C401 is addressable by 05000, 50000, E5000, 5E000, F5001, 5F001.

5.5.4. The possible hex values are 0EEB, 3EEC, 6FEB, 7EEB, 8EEB, ..., FEEB (12 solutions); (2) no solutions; (3) 2F04.

5.5.5. We write the index digit and addressing halfword in the form xbddd. Then (1) 02ABCD may be addressed by 01BAD, 10BAD, 31BB5, 13BB5; (2) 000A4D may be addressed by 00A4D, 03A55, 30A55, 33A5D; (3-5) all of 001139, 88888E, and 02A010 are not addressable.

5.5.6. Even though the base register specification digit would be zero in each addressing halfword, GR0 would still not be used (or usable) as a base register!

5.5.7. (1) X'31C161', X'07CBC5;' (2) X'005CFF', X'2B42A3;' (3) X'023B43', X'000050;' (4) X'000E4E', X'000E4E'.

Section 5.6

5.6.1. Because 15 registers are available for addressing purposes, there is considerably less need to use addresses in memory to locate an operand; the added cost of accessing memory is also avoided. An additional bit would also be required in all instructions for which indirection was allowed. In some processors allowing indirect addressing, it is possible to put the CPU into a loop by having two instructions in an indirect-address chain refer to one another.

Section 6 Solutions

Section 6.2

6.2.1. The default rule is that columns 1 to 15 must be blank. It is a common practice to group comments into blocks of descriptive statements, all of which have an asterisk in column 1. If one of those statements has a nonblank character in column 72, the following record will not be blank in columns 1 to 15, as required.

Section 6.3

6.3.1. The statement will be read by the Assembler during assembly time, when it will be translated into a machine language instruction in the object module. This will be linked into the load module; after the load module has been placed in memory at the start of execution time, the CPU can fetch, decode, and execute the instruction.

6.3.2. The choice is up to you; any column at least one space past the operand field is acceptable.

6.3.3. Any column except 1. It should not extend into column 72!

6.3.4. The operation field is always required.

6.3.5. Comment statements have no operation field entry; some Assembler instructions (such as SPACE, EJECT, START, ORG, CSECT, etc.) require no operand field entry; any statement may be written without a remarks field.

6.3.6. Because column 1 is blank, the operation field entry is LOAD; the operand field entry is therefore “LR”; the rest of the statement is comments!

6.3.7. (a) Only comment statements; (b) All non-comment statements.

Section 6.5

6.5.2. The END statement is only an indication to the Assembler that the *source* module is complete, and it is *not* any part of the executing program. If you understand the phrase “control reaches the END statement” to mean that the CPU attempts to execute instructions outside the bounds of the program, an error is likely to cause the “end” (termination) of program execution. There's no way to know how the CPU will interpret whatever bit patterns it may then “stumble” into.

6.5.3. Lines 4 and 10 have name-field entries; all but line 6 have operation field entries, operand field entries, and comment field entries.

Programming Problem 6.1.

Here is the Assembler's listing of my program:

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement

000000               00000 00038     1 Test   Start 0            First line of program            Line 4
                                     2        Print NoGen                                         Line 5
                                     3 *      Sample Program                                      Line 6
000000 0DF0                          4        BASR  15,0         Establish a base register        Line 7
                R:F  00002           5        Using *,15         Inform the assembler             Line 8
000002 90EF F00A           0000C     6        PRINTOUT  MyName,*  Print name and stop             Line 9
00002A D196889540D94B40            436 MyName DC    C'John R. Ehrman'    Define constant with name  Line 10
000000                             437        END   Test          Last statement                  Line 11
```

When the program was executed, it produced this printed output;

```
*** PRINTOUT REQUESTED AT LOCATION 021002, CC=1
MyName   = 'John R. Ehrman'
*** EXECUTION TERMINATED BY PRINTOUT * AT LOCATION 021002
```

Section 7 Solutions

Section 7.1

7.1.1. (1) 12345 = X'3039'; (2) X'15555'; (3) X'B4DAD'; (4) too long (9 significant digits); (5) the + sign is not valid on a self-defining term; (6) X'3C3C3D'; (7) too long (33 significant bits).

Section 7.2

7.2.1. (1) X'007B7C5B'; (2) incorrectly paired apostrophes; (3) X'40C140C2';
(4) X'00D9E4C4'; (5) X'0000F1F2'; (6) too long (five characters).

7.2.2. The values are (1) X'50' (the ampersands are paired), (2) X'F7F5', (3) X'7D' (the apostrophes are paired), (4) X'C37D', (5) X'F0', (6) X'E2C4E3'.

7.2.3. This table shows the ASCII encodings for the same characters shown in Table 13 on page 87. (Here, they are not in order of increasing encoding values.)

Table 449. ASCII Character Representation. (((alntb7b)))

Char	Hex	Char	Hex	Char	Hex	Char	Hex
Blank	20	.	2E	(	28	+	2B
&	26	$	24	*	2A	)	29
-	2D	/	2F	,	2C	_	5F
#	23	@	40	'	27	=	3D
a	61	b	62	c	63	d	64
e	65	f	66	g	67	h	68
i	69	j	6A	k	6B	l	6C
m	6D	n	6E	o	6F	p	70
q	71	r	72	s	73	t	74
u	75	v	76	w	77	x	78
y	79	z	7A	A	41	B	42
C	43	D	44	E	45	F	46
G	47	H	48	I	49	J	4A
K	4B	L	4C	M	4D	N	4E
O	4F	P	50	Q	51	R	52
S	53	T	54	U	55	V	56
W	57	X	58	Y	59	Z	5A
0	30	1	31	2	32	3	33
4	34	5	35	6	36	7	37
8	38	9	39				

7.2.4. (1) X'7D7D7D', (2) X'3E8', (3) X'8', (4) X'507D50', (5) X'6B', (6) X'C17EC2'.

7.2.5.

(1) 64 = X'40' = B'01000000'= C' ' (in our notation, C'•')
(2) 245 = X'F5' = B'11110101'= C'5'
(3) 80 = X'50' = B'01010000' = C'&&'
(4) 16476 = X'405C'. = B'0100000001011100' = C' *' (in our notation, C'•*')
(5) −101058055 = X'F9F9F9F9' = B'11111001111110011111100111111001' = C'9999'
(6) 12966353 = X'C5D9D1' = B'110001011101100111010001' = C'ERJ'

7.2.6. The bits 11010010 are the EBCDIC representation of the capital letter K.

7.2.7. The terms and their values are:

```
(1) B'110010110000010111010110' = X'00CB05D6'
(2) C'A&&B'                     = X'00C150C2'
(3) 54721                       = X'0000D5C1'
(4) X'B00B00'                   = X'00B00B00'
```

7.2.8. The respective values are:

1. X'D1C5'
2. 53701
3. −11835
4. C'JE'

7.2.9. The values are:

```
1. B'110010111000010111011001' =  X'CB85D9'
2. C'R&&Z'                     =  X'D950E9'
3. 51401                       =  X'C8C9'
```

7.2.10. (1) B'01110101100010' = X'1D62', (2) C'''+' = X'7D4E', (3) 10010 = X'271A'

Section 7.3

7.3.1. Symbols (1), (3), (4), and (9) are valid. The second is invalid if you think of it as a single symbol because the blank is not allowed, but Captain and Major are valid symbols. The other invalid symbols are (5) (exclamation point is not allowed, to say nothing of the language), (6) (starts with a digit), (7) (naturally), (8) (parentheses are not allowed).

7.3.2. Consider the "symbol" 1234567J. The Assembler's term-scanning routine might be well on its way into converting what appeared to be a decimal self-defining term when the letter J appeared. This would require either backing up, or necessitate multiple scans wherever a symbol could appear.

Section 7.5

7.5.1. All the symbols (EX7_5_1, BEGIN, DUMMY, N, and ONE) are relocatable. The LC values (and therefore the values of the symbols) are as shown.

```
 LC     Stmt
5000    EX7_5_1   START X'5000'
5000              BASR  6,0
5002    BEGIN     L     2,N
5006              A     2,ONE
500A              ST    2,N
500E    DUMMY     DS    XL22
5024    N         DC    F'8'
5028    ONE       DC    F'1'
```

Section 7.6

7.6.1. You shouldn't need to check *this* answer!

7.6.2. The symbol TEN is three characters long.

Section 8 Solutions

Section 8.2

8.2.1. The results are A+B, A-B, and A-B.

8.2.2. The expression has value 19.

8.2.3.

```
a.  A+-+-B     Valid    A+(-(+(-B)))    -->   A+B
b.  A*--B      Valid    A*(-(-B))       -->   A*B
c.  A-*-B      Invalid  A-(*(-B))
d.  A---B      Valid    A-(-(-B))       -->   A-B
e.  --A-++B    Valid    -(-A(-(+(+B)))) ..)   A-B
```

Section 8.3

8.3.1. If none of these expressions is further combined with other expressions, then A−R is invalid because the only relocatable term is preceded by a − sign. (If it was combined with another expression such as in (A−R)+(A+R) the result would be valid and absolute. (It's important to look at the whole expression!)

R+R is invalid because the sum will be complexly relocatable; all the expressions involving multiplication or division with a relocatable term are always invalid; the remaining expressions are always valid.

8.3.2. If you think so, you may still be uncertain about the difference between the Assembler's computation of expression values at assembly time, and the program's computation of whatever it likes at execution time. A program *can* compute remainders, as we will see when we discuss the Divide instructions.

8.3.3. The two symbols may have different relocatability attributes. (We will meet such symbols when we discuss external symbols and control sections.)

Section 8.4

8.4.1. 75, 32, and 2400 respectively.

8.4.2. R6 is the only absolute symbol, and it has value X'000009' (a poor practice). The other symbols are relocatable; their values are A (X'000466'), B (X'00046C'), C (X'000470'),
X (X'000474'). The expressions, their values, and their relocatability attributes, are:

```
B+X'1C'            X'000488'      relocatable
C-A+X-2*(R6/2)     X'000476'      relocatable
2*C'-'-C'A'+2      X'000001'      absolute
B-2                X'00046A'      relocatable
R6-2               X'000007'      absolute
```

8.4.3. Expressions 1 and 6 are relocatable; the rest are absolute. They have values X'10A99', X'FFFFE884', X'7', X'3', X'8D', and X'12098' respectively.

8.4.4. Item 4 is invalid; if you look at it long enough you will see why. Expressions 1, 3, 5, 6, and 7 are valid. Expressions 3 and 5 are syntactically valid, but the value overflows a word. Only the rightmost 32 bits would be retained.

8.4.5. (1) X'000F08', absolute; (2) X'172BE9', relocatable; (3) X'001BD8', absolute;
(4) X'000180', absolute; (5) X'173ACC', relocatable.

8.4.6. Expressions 1 and 5 are invalid, because they contain products involving a relocatable expression. Except for expression 4, the other expressions are all absolute, and have the following values: (2) X'00558F'; (3) X'0020A9' (if you got X'0020A8', you forgot to do the multiplication before the division); (4) X'00142D0A' (relocatable); (6) X'FFFFFFFE' (–2 in decimal).

8.4.7. If we allowed relocatable terms in multiplications and divisions, it would no longer be possible to preserve the simple additive relationship between the location value of an assembly-time expression and its address value at execution time. Handling such constructions would require that we pass enough information about the expression to the Program Loader so that it could compute the correct value when the true address referred to by the symbol is known.

8.4.8.

```
1. B-A      Value=X'F08', absolute
2. A+C'.'   Value=X'172BE9', relocatable
```

Section 8.5

8.5.1. (1) second; (2) first; (3) second; (4) second; (5) first; (6) third; (7) third; (8) third, (9) second. All the operands are valid!

Programming Problem 8.1.

Many of the Length Attribute References I tried were invalid. The reasons are explained among the lines of the Assembler's listing.

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement

000000                00000 00024     1 P8_1    Start 0
                 R:F  00000           2         Using *,15
                      001A9           3 ABS425  Equ   425
000000 0000 0000            00000     4         LA    0,L'2
** ASMA147E Symbol too long, or first character not a letter - 2
```

The Length Attribute of a self-defining term is undefined; if it had been written `X'2'`, the Assembler would produce a different error message.

```
000004 4100 0001            00001     5         LA    0,L'ABS425
** ASMA019W Length of EQUated symbol ABS425 undefined; default=1
```

The Length Attribute of the absolute symbol is undefined, so the Assembler tells us that it has assigned a default length attribute, 1.

```
000008 4100 0004            00004     6         LA    0,L'*
```

The Length Attribute of a Location Counter Reference is valid. The instruction is 4 bytes long, so that's the value of the Length Attribute Reference.

```
00000C 0000 0000            00000     7         LA    0,L'*-10
** ASMA028E Invalid displacement
```

The expression `L'*-10` is not addressable, because while `L'*` has value 4, `L'*-10` has value –6. The Assembler could not create a valid addressing halfword.

```
000010 0000 0000            00000     8         LA    0,L'(*-10)
** ASMA147E Symbol too long, or first character not a letter - (*-10)
```

The Length Attribute of a parenthesized expression is undefined.

```
000014 4100 0004            00004     9         LA    0,L'=F'10'
```

The Length Attribute of a literal is well defined.

```
000018 0000 0000            00000    10         LA    0,L'L'*
** ASMA150E Symbol has non-alphanumeric character or invalid delimiter - L'*
```

The Length Attribute of a Length Attribute Reference is not allowed.

```
                                     11         End   P8_1
000020 0000000A                      12                =F'10'
```

Programming Problem 8.2.

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement

                      00005           1 A      Equ   5
                      00003           2 B      Equ   3
                      00003           3 C1     Equ   L'A*B
** ASMA019W Length of EQUated symbol A undefined; default=1
                      00005           4 C2     Equ   A*L'B
** ASMA019W Length of EQUated symbol B undefined; default=1
```

The Length Attribute of the symbols A and B is undefined, so the Assembler tells us that it has assigned a default length attribute, 1.

```
                                         5 C3    Equu   L'(A*B)
** ASMA147E Symbol too long, or first character not a letter - (A*B)
** ASMA158E Operand expression is defective; set to *
```

A Length Attribute Reference to an expression is undefined. In this case, the Assembler tells us that it has assigned a default value, the Length Attribute of the current Location Counter.

Section 9 Solutions

Section 9.2

9.2.1. The values of the expressions are (1) 0, (2) 13, (3) 17, (4) 15, and (5) 16. This means that (3) and (5) are invalid.

9.2.2. Only (a) is valid; (e) is even, but too large.

Section 9.5

9.5.1. The operand $D_2(X_2,B_2)$ is of the form expr(expr,expr); $D_2(,B_2)$ is of the form expr(,expr); $S_2(X_2)$ is of the form expr(expr); S_2 is of the form expr.

9.5.2. (1) implied, no indexing; (2) implied, with indexing; (3) implied, with indexing; (4) explicit, with indexing; (5) implied, with indexing; (6) explicit, no indexing.

9.5.3. (1) implied (invalid if A and B are both relocatable); (2) implied (invalid if both A and B are relocatable); (3) implied, with indexing (but invalid, since `C')'` has value greater than 15); (4) implied, with indexing (but invalid, since `C',C'` exceeds 15); (5) implied; (6) explicit, with indexing (but invalid, since the X_2 and B_2 expressions are both too large); (7) explicit, with an index expression); (8) explicit, no indexing.

Section 9.7

9.7.1. The implied addresses are S_1 and S_2.

(1) Can be only SI: $1=D_1$, $2=B_1$, $3=I_2$.
(2) Can be only RS-1: $4=R_1$, $5=D_2$, $6=B_2$.
(3) Can be only RS-2: $7=R_1$, $8=R_3$, $9=S_2$.
(4) Can be either RS-1: $10=R_1$, $11=S_2$; or SI: $10=S_1$, $11=I_2$.
(5) Can be only RS-1: $14=R_1$, $15=D_2$, but 16 is not a valid value for B_2.
(6) Can be only SI: $100=S_1$, $101=I_2$.

Section 9.9

9.9.1. In the following table, the headings "SS-1" and "SS-2" refer to the two SS-type instruction formats.

	Operand	SS-1 Format	Length	SS-2 Format	Length
(1)	`1(2)`	$S_1(N_1)$ or $D_2(B_2)$	explicit	$S_1(N_1)$ or $S_2(N_2)$	explicit
(2)	`4(5,6)`	$D_1(N_1,B_1)$	explicit	$D_1(N_1,B_1)$ or $D_2(N_2,B_2)$	explicit
(3)	`A(L'B)`	$S_1(N_1)$ or $D_2(B_2)$	explicit	$S_1(N_1)$, $D_2(B_2)$, $S_2(N_2)$ or $D_2(B_2)$	explicit
(4)	`Line`	S_1, S_2	implicit	S_1, S_2	implicit
(5)	`Line(80)`	$S_1(N_1)$	explicit	Invalid!	
(6)	`XX(,5)`	$D_1(,B_1)$	implicit	$D_1(,B_1)$ or $D_2(,B_2)$	implicit

Note:

(c) The form D(B) is valid only if `A` is absolute and `L'B` is between 0 and 15; but it is very unlikely anyone would specify B_2 that way!
(e) 80 cannot be either a length or a base register for an SS-2-type instruction.
(f) The symbol `XX` must be absolute.

Section 10 Solutions

Section 10.5

10.5.1. The two statements are in reversed order. The value of the LC *before* the BASR is encountered may be 2 (or even 3) less than the value of the LC after the BASR has been assembled. Thus the value placed in the USING Table by the Assembler will cause it to calculate displacements that are 2 (or 3) bytes too large. This will undoubtedly lead to incorrect operand addresses when the program is executed.

Stated differently: the value of the USING base_location (at assembly time) relative to the start of the program will not be the same as the base address (at execution time), relative to the start of the program.

Section 10.6

10.6.1. If by chance the contents of the word at `N` had been the decimal integer 20450=`X'4FE2'` instead of 8, then the Effective Address of the A instruction would be `X'4FE2'`+`X'26'`=`X'5008'`, a perfectly acceptable memory address for a word (and, besides, it's somewhere inside our program!). The subsequent instructions would have proceeded blindly, adding the contents of the word at memory location `X'5008'` (portions of the A and ST instructions!) to the *unknown* contents of register 2, and storing the result at location `X'5004'`.

Following execution of the ST instruction, the first portion of the program segment would then contain

```
0D60 5860 xxxx xxxx 6026 5020 6022 ...
```

where `xxxx xxxx` is whatever sum resulted in register 2. The constants named `N` and `ONE` would be unchanged.

Section 10.7

10.7.1. The object code should be

```
0DA0
41D0A12A
4110A172
450EA19E
500FA176
```

10.8.1. The object code should be

```
5830B064
4A30B060
1043
9034B058
4240B056
4770B02A
```

Section 10.10

10.10.1. The statements are syntactically valid. However, the DROP statement should refer to the number of a general register, which must be between 0 and 15. The resulting diagnostic will likely say something about an invalid register number.

10.10.2. Following the first USING, the USING Table contains

basereg	base location	RA
9	00004002	01

Following the second USING, the new entry in the USING Table will be

basereg	base location	RA
A	00004008	01

The DROP 9 eliminates the first entry, and the DROP 10 eliminates the second. The generated code is as follows:

```
Loc    Stmt             Generated Code

4000   BASR  9,0        0D90
4002   USING *,9        none
4002   L     4,*+54     58409036
4006   BASR  10,0       0DA0
4008   USING *,10       none
4008   L     3,*+52     5830A034  (GPR 10 has a smaller displacement)
400C   DROP  9          none
400C   L     2,*+48     5820A034
4010   DROP  10         none
4010   L     1,10(,9)   5810900A
```

Because register 9 was unchanged when the last instruction was executed, it can still be used by the CPU to calculate effective addresses. Thus, the word at X'400C' will be loaded into register 1; its contents will therefore be X'5820A034'.

The result does not depend on where the instructions were loaded into memory.

Section 10.11

10.11.1. The symbol at B is not addressable by the L instruction, because a displacement −2 would be required. Because HLASM normally zeros instructions with serious errors (like this one), execution would probably fail with a program interruption for an invalid operation code. This may help show why it's important to fix all assembly errors before execution.

Section 10.12

10.12.1. This is generally desirable; an implied address such as X'356' in a statement should lead to an effective address of the same value. If the Assembler did not automatically supply a zero base digit, we would have to write machine instruction operands in the form X'356'(0,0) or XX'356'(0) which is less convenient, although it is more explicit.

Programming Problem 10.1

Here is a source program:

```
         TITLE 'SOLUTION TO PROBLEM 10.1'
P10_1    START   X'5000'
         BASR    6,0
         USING   BEGIN,6
BEGIN    L       2,N
         A       2,ONE
         ST      2,N
         DS      22X
N        DC      F'8'
ONE      DC      F'1'
         END     P10_1
```

The assembly listing looks like this:

```
005000                05000 0002C     2 P10_1    START   X'5000'
005000 0D60                           3          BASR    6,0
                 R:6  05002           4          USING   BEGIN,6
005002 5820 6022            05024     5 BEGIN    L       2,N
005006 5A20 6026            05028     6          A       2,ONE
00500A 5020 6022            05024     7          ST      2,N
00500E                                8          DS      22X
005024 00000008                       9 N        DC      F'8'
005028 00000001                      10 ONE      DC      F'1'
005000                               11          END     P10_1
```

The addressing halfwords of the instructions in statements 5, 6, and 7 are the same as those we calculated by hand.

Section 11 Solutions

Section 11.1

11.1.1. Because System z supports a wide variety of data types, it would be difficult to know what conversion should be used to convert “8” to the correct internal representation. (Some assemblers use a different instruction mnemonic to indicate the desired type of internal representation.)

Section 11.4

11.4.1. The generated constants are

```
(1) X'81'
(2) X'0080'
(3) X'FFFF9D000063'
(4) X'7F'
```

11.4.2. The alignments are byte, halfword, and word respectively.

Section 11.6

11.6.1. He defined four word constants, having values 1, 0, 0, and 0. (The four generated words contain the value 2^{96} as a 128-bit binary constant, which might be useful to someone doing 128-bit binary arithmetic.)

He could either rewrite the constant without the commas (and learn to count carefully), or use blanks, as in

```
TEN_to_9 DC    F'1 000 000 000'
```

The resulting constant is X'3B9ACA00'.

11.6.2. The generated constant is `X'00000001 FFFFFFFF 00000001 FFFFFFFF'` (where spaces have been inserted for readability).

Section 11.8.

11.8.1. The four values are

```
(1)      F'20000'   or  F'2E4'        X'00004E20'    Aligned
(2)      F'50000'   or  F'5E4'        X'0000C350'    Aligned
(3)      FL2'20000' or  FL2'2E4'      X'2710'        Not aligned
(4)      FL4'80'    or  FL4'8E1'      X'00000050'    Not aligned
```

11.8.2. Consider these two constants:

```
Ten_to_9 DC    F'1E9'
Ten_to_9 DC    FE9'1'
```

11.8.3. The object code is X'5820F036, 5A20F03A, 5B20F03E, 5020F042'. The four missing LC values are X'000038, 00003C, 000040, 000044'.

Section 12 Solutions

Section 12.1

12.1.1. Try F'1E9'.

12.1.2. The first constant will generate X'00000001'; all the others generate zero.

12.1.3. The Assembler rounds the fraction part — the bits lost at the right of the radix point — so the generated constant is X'0001'. (If you use a scale or exponent modifier, you can create fixed-point binary constants with fractional parts. We'll see some examples later.)

12.1.4. The generated constant will be X'80000000', with an Assembler error message saying the value is too large. The operand is treated as a *signed* value, and exceeds $2^{31}-1$ by 1.

12.1.5. The two binary integer constant words (named N and ONE) must be on a word boundary, so their addresses will increase by 4 (not 2!) to X'5028' and X'502C'.

12.1.6. The values are:

```
         DC    F'-2147483620'     X'8000001C'
         DC    H'-32594'          X'80AE'
         DC    F'+2147483260'     X'7FFFFE7C'
```

Section 12.2

12.2.1. The values are X'0000026C', X'00000564', and X'00000012'.

Section 12.3

12.3.1. The values are X'00C1', X'0B5E', and X'002D'.

12.3.2. S-constants 1, 3, 4, and 5 are valid, but S(A(N)) is not, since the displacement A is not absolute. Constants 1 and 5 depend on USING information, but normally there will be no USING statement that would affect S(N). In S(N(7)), N must have a value between 0 and 4095, and in S(7(N)), N must have a value between 0 and 15.

Section 12.4

12.4.1. 33, 7, and 5 bytes respectively. (Did you peek at Section 12.5?)

12.4.2. You'd really be doing it the hard way this way. Since the implied length is 100 bytes, you must write between 793 and 800 binary digits. Assuming you write 54 per record (using the usual continuation rules), you would need 15 records, which could exceed the allowable maximum for your Assembler. There may be a better way.

12.4.3. You can't; think about it again. (In a character constant, a comma is part of the nominal value, not a separator.)

12.4.4. Two possible interpretations, and their Assembler Language defining statements, are:

```
         DC    F'1077952604'    Word binary integers
         DC    C'•••*'          Characters (three blanks and an asterisk)
```

Other interpretations are possible, as we'll see later.

12.4.5. The value of four blank characters interpreted as an integer is 1,077,952,576.

12.4.6. The generated constants are:

(1) X'F1'
(2) X'001F'
(3) X'00123456'

12.4.7. The generated constants are

1. X'C17DC250C3'
2. X'7DC150C27D7DC3'
3. X'C1C2C3C67D'

12.4.8. Just write

```
EBCHex   DC    C'0123456789ABCDEF'
```

12.4.9. The generated constants for each symbol, and the differences, are:

```
Symbol  Generated Constant  Differences
A       X'0000000000'       Both constants generate the same data,
B       X'0000000000'       A and B have different length attributes.

C       X'0707070707'       Different data is generated, and
D       X'0000000007'       C and D have different length attributes.

E       X'4040404040'       Both constants generate the same data,
F       X'4040404040'       E and F have different length attributes.

G       X'5C5C5C5C5C'       Different data is generated, and
H       X'5C40404040'       G and H have different length attributes.
```

12.4.11. X'F3F4F540'.

Section 12.5

12.5.1. The DC operand CL2'ABC' is truncated on the right, so the generated constant will be X'C1C2'. In the address constant, the value of the expression is X'00C1C2C3', which will be truncated on the *left*, giving X'C2C3' for the generated constant.

12.5.2. The constants and their alignments are

```
(1) F'1000'     X'000003E8'     Word
(2) H'1000'     X'03E8'         Halfword
(3) B'1000'     X'08'           Byte
(4) XL1'1000'   X'00'           Byte (truncated)
(5) CL1'1000'   X'F1'           Byte (truncated)
(6) AL1(1000)   X'E8'           Byte (the last 8 bits of the term)
(7) YL3(1000)   Length error
```

The constant AL1(1000) does not generate an error message!

12.5.3. The values are:

```
(1) X'00F1'
(2) X'01E2'
(3) X'001234'
(4) X'2345'
```

12.5.4. The constants that cannot be fit into smaller fields are X'56789' and Y(X'124').

Section 12.6

12.6.1. The generated constant is X'000005000002'.

12.6.2. The generated constant is X'C2D3C1D5D25040'.

12.6.4. If the constant named Message starts on a word boundary, it will end one byte past a word boundary, so that three bytes must be skipped to align the A-type constant named MsgLen. If it starts at a location one byte before a word boundary, zero bytes must be skipped. Thus, anywhere between zero and three bytes will be skipped.

Section 13 Solutions

Section 13.1

13.1.1. Nothing is generated, since they are all DS statements. The length attribute of each symbol is 4, with the length of Y being implied. The values of the symbols are X'12345', X'1234C' (note alignment), and X'12350' respectively.

Section 13.2

13.2.2. The solutions are shown in this table:

Sym	Value	LA
J	X'346'	2
K	X'34C'	1
L	X'350'	4
P	X'345'	3
Q	X'348'	3
R	X'350'	4
T	X'345'	2
V	X'348'	2
W	X'350'	4

13.2.3. The values and Length Attributes are:

```
1. A       DC    F'2'               A has value X'348', Length Attr. = 4

2.         DS    0H
           DC    C'*'
   A       DC    C'Asterisk'        Value=X'347', Length=8

3.         DC    0F'1'
   A       DC    0XL27'0'           Value=X'348', Length=27

4. A       DC    A(A)               Value=X'348', Length=4

5.         DS    19H
   A       DC    X'12345'           Value=X'36C', Length=3

6.         DC    3CL4'ABCDE'
   A       DC    C'A&&B'            Value=X'351', Length=3

7.         DS    CL400
   A       DC    F'12,34,56'        Value=X'4D8', Length=4
```

Section 13.3

13.3.1. The length attribute of a symbol is determined from the length attribute of the first term in the first-operand expression. In this case, that term is the symbol Table, which has length attribute 4.

13.3.2. The result will be incorrect only if the length attribute of the symbol HW8 is expected to be 2. This is considered a poor programming practice. It runs the risk that someone might change the constant named **FW8** to some other value with another name, and then the symbol **HW8** could be undefined!

13.3.3. The first two definitions are equivalent, and the last two are equivalent (the parentheses have no effect other than improving readability). The first two definitions, however, are incorrect: if the value of TblSiz is odd, the value assigned to MidTbl will be aligned on a *halfword* boundary *between* two word boundaries, and attempts to refer to the word at MidTbl would be erroneous.

13.3.4. (See your solution to Exercise 11.7.1 also.) The symbols A3 and A5 behave in exactly the same way as the symbols they are equated to. Symbol A4 will not have the expected value, because the Assembler does not allow literals as operands of EQU statements. (What advantages would there be to allowing literals as EQU operands?)

13.3.6. The definitions are circular, so the Assembler can't determine what should be done.

13.3.7. Both pairs are valid. (For some very old assemblers, the second pair would cause the symbol BAKER to be undefined because its operand was not yet defined.)

13.3.8. All symbols are relocatable. The symbols, their values, and their length attributes, are:

```
Symbol   Value   Length
ST       01DBC5    8
W        01DBC8    4
X        01DBD0    4

P        01DBC8    4
Q        01DBC8    2
R        01DBC8    1
S        01DBCC    1
```

13.3.9. You will remember from Section 8.3 that the assembler evaluates division by zero in an expression as zero. First, we show how to break the calculation into four steps, using four auxiliary symbols **K1**, **K2**, **K3**, and **K4**.

```
K1        Equu  A/B                 0 if A < B, > 0 otherwise
K2        Equ   B/A                 0 if B < A, > 0 otherwise
K3        Equ   K1/K2               0 if A < B,   1 otherwise
K4        Equ   K2/K1               0 if B < A,   1 otherwise
MaxOfA_B EQU    A*K3*(1-K4)+B*K4    Final result
```

The factor (1-K4) is needed in case A and B are equal. Then, we can write the full expression as follows:

```
MaxOfA_B EQU   ((A/B)/(A/B))*A+(1-((A/B)/(A/B)))*((B/A)/(B/A))*B
```

It's not very pretty, but it works!

13.3.10. The values and Length Attributes are:

1. Value = X'2924', Length attribute = 4
2. Value = X'291B', Length attribute = 3
3. Value = X'0009', Length attribute = 2

13.3.11. The symbols and their value and Length Attributes are:

```
1. A        DS     F                     Value=X'12348', Length=4
   B        DS     2H                    Value=X'1234C', Length=2
   C        DS     2CL2                  Value=X'12350', Length=2

2. F        DC     A(F)                  Value=X'12348', Length=4
   G        DC     3AL3(F,G,H)           Value=X'1234C', Length=3
   H        DC     Y(*-F,275)            Value=X'12368', Length=2

3. P        DC     2C'3&&'               Value=X'12345', Length=2
   Q        DC     2A(C'3&&')            Value=X'1234C', Length=4
   R        DS     3XL3'FEDCBA93'        Value=X'12354', Length=3

4. X        DC     0FL5'5,10,20'         Value=X'12345', Length=5
   Y        DC     FL3'5,10,20'          Value=X'12345', Length=3
   Z        DC     2C'5,10,20'           Value=X'1234E', Length=7
```

13.3.12. The symbols and their generated data are:

- F   X'00012348'
- G   X'012348 01234C 012368' (repeated 2 more times!)
- H   X'00200113'
- P   X'F350F350'
- Q   X'0000F3500000F350'
- Y   X'00000500000A000014'

13.3.13. The symbols and their values and Length Attributes are:

```
1. STR      DS     0CL8                  Value=X'1DBC5',  Length=8
   W        DS     2F                    Value=X'1DBC8',  Length=4
   X        DS     2F                    Value=X'1DBD0',  Length=4

2. P        DS     0F                    Value=X'1DBC8',  Length=4
   Q        DS     0H                    Value=X'1DBC8',  Length=2
   R        DC     4X'40'                Value=X'1DBC8',  Length=1
   S        EQU    *-P                   Value=X'00004',  Length=1
```

13.3.14. The values of the symbols, the length attributes, the data locations, the final LC value, and the assembled data are given in hex. Underlined zeros indicate padding bytes inserted by the Assembler.

	Symbol	Value	L.Att	DataLoc	LCval	data
1.	A	128	4	128		000000FFFFFFEF
	B	12C	2	12C	12E	0021
2.	D	125	4	125		00000011
	C	12A	2	12A	12C	00FFDF
3.	E	125	8	125		C1C2C3C4C5C6C7C8
	F	130	4	130	134	000000000003E8
4.	G	126	2			
	H	12C	4	12C	130	0000129E
5.	J	126	2			
	K	126	1			
	L	128	4			
	M	128	6			
	N	128	4	128	12C	FFFFFC18
6.	P	125	3	125		C17DC2C17DC2C17DC2
	Q	130	4	130	138	000000C17DC200C17DC2
7.	R	128	4			
	S	2B8	50		308	
8.	AB	128	4	128	134	00000000000384 0080000002 000001
9.	BC	125	2	125		00070007000701
	CD	12C	2	12C	130	3FFFC001
10.	DE	128	4			
	EF	13C	4		13C	
11.	T	125	2	125		0CAB
	V	127	2	127		015C015C
	W	12B	8	12B	13B	C3C1C24040404040C3C1C24040404040
12.	Y	126	2			
	X	128	2	128	12C	00020002
13.	Z	125	2	125		E9E9
	ZZ	128	4	128	130	000000000300000003000000003

13.3.15.

```
REven    Equ   ((N+1)/2)*2        In case N is odd!
ROdd     Equ   REven+1            Next higher odd number
```

13.3.16. In the last three cases (11-13) at least one expression depends on the value of a symbol being defined by the same statement that is using it. For many early assemblers this required making more complex decisions than could be justified at that time. All 13 cases are successfully resolved by the High Level Assembler.

13.3.17. Consider

```
E        Equ   ((1-A)/(1+A))           Explain why it works!
```

13.3.18. Consider

```
E        Equ   (A/A)                   Explain why it works!
```

What happens if **A** has negative values?

Section 13.4

13.4.1. The length and value attributes are:

Symbol	Length	Value
Cost	8	X'2024'
Desc	60	X'202C'
Fill	29	X'2068'
LFill	1	X'201D'
Pfx	24	X'2000'
Prod	12	X'2018'
Result	133	X'2000'

Section 13.5

13.5.1. This technique will work correctly so long as nothing is ever stored at either FW8 or HW8. Another way to get the same result is to write

```
FW8      DC    F'8'
HW8      Equ   FW8+2,2            Define halfword, length attribute 2
```

13.5.2. The asterisk in column 1 indicates a comment, not the value of the LC! The last statement should be replaced by ORG Here.

13.5.3. (1) B has value X'9830', and length attribute 2. (2) B has value X'9838', and length attribute 8. (3) B has value X'9860', and length attribute 4.

13.5.4. (1) B is relocatable, has value X'9837', and length attribute 5. (2) B is relocatable, has value X'B942', and length attribute 1.

13.5.5. We could write

```
F1       DC    F'1'         First constant
X2       DC    X'2'         Second constant
*                           Three zero (skipped) bytes here
F3       DC    F'3'         Third constant
         ORG   X2+1
SKIP3    DS    XL3          Name and length of skipped bytes
```

Using Extended EQU Syntax, we could replace the last two statements:

```
SKIP3    Equ   X2+1,3       Name and length of skipped bytes
```

13.5.6. The symbol SET will be the operand of the ORG instruction, which is undoubtedly not what was intended. If a symbol SET has been defined somewhere in the program (which is just a matter of chance), the LC will be assigned its value, but that is quite unlikely to be the highest LC value attained in the program. (Or what is wanted!)

13.5.7. The assigned and desired lengths are both 80 bytes.

For the “Bonus” question, you could write something like

```
         DS    (80-StmtLen)XL(StmtLen-80+1)
```

If StmtLen has exactly the desired value 80, this statement becomes

```
         DS    0XL1
```

but if its value is greater than 80 the duplication factor will be negative; and if its value is less than 80, the explicit length will not be greater than zero. In both cases, the Assembler will issue an error message.

Section 13.6

13.6.1. The storage definitions are the same, but the length attributes of the symbol FGroup are different. (What are their values?) One way to see the similarity of the definitions is to consider defining

```
NWords   Equ   1                  Table has one word
```

Both definitions will allocate both Table and LastWord at the same location.

Section 13.7

13.7.1. The Assembler doesn't rescan LC-dependent expressions unless they appear as nominal values in A-type or Y-type constants. The statements will generate ten copies of the first constant, X'00000001'. (It would be helpful if the Assembler indicated that the operand has a modifier requiring re-scanning if a Location Counter Reference appears in the expression.)

13.7.2. The solutions are shown in this table:

Sym	Value	LA	Generated constant
A	X'12345'	3	X'E450C9'
B	X'12348'	4	X'00012348'
C	X'12346'	2	X'0089'
D	X'12345'	3	X'00020002'
E	X'12345'	4	X'C14D405D**000000**00000040'
F	X'12345'	1	X'**00**00010003'

The underscored bytes in the generated constant for E are zero bytes inserted for alignment in the middle of the constant. Whether or not the underscored bytes for F are generated depends on whether the immediately preceding field contains just-generated bytes (in which case the padding byte is present).

13.7.3. The expression (*-Sqrs) increases by 2 each time a constant is generated; because the table starts at one-squared, we add 2. Then, because the product increases by 4 for each constant, we divide by 4.

13.7.4. Your assembled table should look like this (use the Assembler's PRINT DATA statement to see all the generated data):

```
0001010201020203 0102020302030304
0102020302030304 0203030403040405  <-- Note the '03' byte
0102020302030304 0203030403040405
0203030403040405 0304040504050506
0102020302030304 0203030403040405
0203030403040405 0304040504050506
0203030403040405 0304040504050506
0304040504050506 0405050605060607
0102020302030304 0203030403040405
0203030403040405 0304040504050506
0203030403040405 0304040504050506
0304040504050506 0405050605060607
0203030403040405 0304040504050506
0304040504050506 0405050605060607
0304040504050506 0405050605060607
0405050605060607 0506060706070708
```

To see how the DC statement works, consider the underscored byte at offset 19. The first term is *-T, or 19. The next term subtracts (*-T)/2, or 9. The next term subtracts (*-T)/4, or 4. The fourth term subtracts (*-T)/8, or 2, and the fifth term subtracts 19/16, or 1. All succeeding terms subtract zero. Thus the final value is 19-9-4-2-1, or 3. The binary representation of 19 is B'10011' which has three 3 1-bits.

Programming Problem 13.2.

You can write your program this way:

```
P13_2    Start 0
         Using *,15
         Printout Chars,*
         DS    0F                Align to fullword boundary
Chars    DS    0CL16
Ints     DC    F'-1046306171,-1803381883,-1723823710,1082565781'
         End   P13_2
```

You should believe the printed result!

```
 Chars    = C'Assembler is fun'
```

Section 14 Solutions

Section 14.1

14.1.1. The first pair of instructions will load the word *from memory* at address X'00000008' into GR5 (and who knows what that will be?) The second will load a word integer constant X'00000008' into GR5. The distinction here is between the value of an addressing expression, and the value of the object in memory referenced by an addressing expression.

Section 14.2

14.2.1. There is no difference in their actions. However, the ST instruction can be indexed, while STM cannot.

14.2.2. The first word of an area of memory occupied by successive words has displacement 0 from the start of the area; thus the fourth word has displacement 12.

14.2.3. All sixteen registers can be modified by an appropriate LM instruction.

14.2.4. (1) c(X) replaces c(GR15); (2) c(GR0) replaces c(X); (3) c(X) replaces c(GR0). In each case, only one register participates.

14.2.5. Let **A** be the R_1 register expression, and **B** be the R_3 register expression. Then **NREGS** is defined by

```
NREGS    EQU    (B-A)+17*(A/A)-((A-1)/(A+1))-16*((B/A)/(B/A))
```

The value is the sum of four expressions: the first parenthesized expression is simply the usual difference; the second is added if **A** is nonzero; the third is nonzero only when **A** is zero; and the last expression is added when **B** is greater than or equal to **A**. Try some actual examples to see how it works.

Section 14.3

14.3.1. This form of the STH instruction would produce a bit pattern with the correct sign and the 15 low-order bits of the true 32-bit representation. For example:

```
X'00010001'  when stored would become  X'0001'
X'0000FFFF'  when stored would become  X'7FFF'
X'FFFF0001'  when stored would become  X'8001'
```

so there would be no improvement; use halfword data carefully! To be truly useful, such “STH” instructions should indicate an overflow condition if the sign bit was not identical to the next 16 bits to its immediate right.

14.3.2. After the first execution, c(GR2)=X'00005678' After the second, c(GR2)=X'FFFFBA98'.

14.3.3. It could be either. These three statements generate the same object code:

```
STH   4,X'05C'(0,4)
DC    C'   *'              3 spaces and an asterisk
DC    F'1077952604'        4-byte binary integer
```

14.3.4. All the “L” opcodes end in 8, and all the “ST” opcodes end in 0. Similarly, the L/ST, LH/STH, and LM/STM pairs each start with the same first hex digit.

Section 14.4

14.4.1. A useful solution is to construct a table of bytes, each containing the count of 1-bits in the difference between its location and the location of the beginning of the table. Thus, we would define the constant

```
*                  0 1 2 3 4 5 6 7 8 9 ...      15 16 ...  (value of byte at XX)
NBits    DC    AL1(0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,...)  (number of 1-bits)
```

and use it as follows:

```
L     1,=F'0'       Set GR1 to zero for IC instruction
IC    1,XX          Get byte at XX with bits to count
IC    0,NBits(1)    Use it as index into NBits table
STC   0,XX          Replace byte at XX by its bit count
```

Section 14.5

14.5.1. These two instructions and a two-byte temporary memory area will do the job:

```
         STCM  1,B'0110',Temp
         ICM   1,B'0110',Temp
         - - -
Temp     DS    XL2
```

Section 14.6

14.6.1. The Assembler could simply interpret

```
         STR   R1,R2
as
         LR    R2,R1
```

This can be done easily with a macro-instruction, and no changes to the Assembler would be needed.

14.6.2. For LR, LTR, and LNR, the CC will never be set to 3 for any operand in GR R_2. For LCR and LPR, the CC will be set to 3 if the GR R_2 operand is the maximum negative number X'80000000' or −2147483648.

Section 14.7

14.7.1. There is no STGH instruction because its function would be exactly the same as STH.

14.7.2. Here is a solution using Load and Store instructions:

```
         STG   0,DSave           Save all of GR0
         L     0,DSave           Load high-order half into low-order 32 bits
         LMH   0,0,DSave+4       Load low-order half into high-order 32 bits
         - - -
DSave    DS    D                 Temporary storage
```

There are easier ways to do this, as we'll see when we examine the shift instructions.

Section 14.8

14.8.1. Suppose GR2 contains X'xxxxFEDC', where the rightmost 4 digits are a valid halfword integer and the "xxxx" are unwanted bits. Writing

```
         LHR   2,2
```

will extend the sign, leaving X'FFFFFEDC' in GR2.

14.8.2. After the first execution, c(GR2)=X'00005678' After the second, c(GR2)=X'FFFFBA98'.

Section 14.9

14.9.1. Indexing applies only to the RX- and RXY-type instructions L and LG, but the result is equivalent to using LT and LTG, both of which can be indexed.

14.9.1. None.

Section 14.10

14.10.1. The third digit of the opcode is 0 for the RRE-type instructions that deal with all 64 bits of a general register, and 1 for the equivalent instructions that extend 32-bit data to 64 bits.

14.10.2. Complementing before sign extension could produce incorrect overflow indications. Remember that all 32-bit binary numbers can be complemented in a 64-bit representation without overflow. Thus, the source operand X'80000000' (-2^{31}) would produce an overflow when complemented.

14.10.3. Complementing a 64-bit operand can cause overflow, but after extending a 32-bit operand to 64 bits, complementation cannot cause overflow.

14.10.4. The contents of general register 0 and the CC settings are:

```
(1)  X'........ 80000000',  CC=3 (overflow), high-order 32 bits unchanged
(2)  X'FFFFFFFF 80000000',  CC is unchanged
(3)  X'00000000 80000000',  CC=2 (positive)
(4)  X'FFFFFFFF 80000000',  CC=1 (negative)
```

14.10.5. All 32-bit integer complements can be correctly represented in 64 bits.

Section 14.11

14.11.1. Load zeros into the R_1 register, and then insert the desired byte using an IC instruction.

14.11.1. Load zeros into the R_1 register, and then load the desired word using a L instruction.

Section 15 Solutions

Section 15.2

15.2.1. The value of the CC is exactly the offset of the tested bit in the M_1 field of the instruction.

15.2.2. The CC must take one of the values 0, 1, 2, or 3, each of which corresponds to a 1-bit in the mask. The branch condition is always met.

15.2.3. The next instruction to be fetched will come from memory address zero, which (very!) rarely contains your program's instructions.

15.2.4. If n = 0, or the branch condition is not met, nothing happens. If n ≠ 0 the Effective Address is n+c(GRn)+c(GRn). Then, if n is even and the branch condition is met, there could be an error if the Effective Address is too large or is the address of a non-instruction. If n is odd, a specification error would occur because the Effective Address is odd.

Section 15.4

15.4.1. The given code sequence correctly aligns the address constant on a word boundary immediately preceded by the BASR. However, the DS 0F may have caused one, two, or three bytes to be skipped *preceding* the NOPR!

15.4.2. There are many possibilities; you only need 2-byte or 4-byte instructions that do not branch "out of line" or change the CC. For example, in place of NOPR you could use `LR 0,0`; and in place of NOP you could use `ICM 0,0,0` or `LGR 0,0` or `BC 15,*+4`. (Operands like `*+4` are considered poor programming practice.)

NOPR and NOP are preferred, because the CPU need not "move" data uselessly. For NOPR, the CPU can determine from the zero branch mask that it need not consider alternative targets for the next instruction to be fetched, which it must do for `BC 15,*+4`.[328]

15.4.3. No branch will occur, because the R_2 digit is zero; this was described in Section 15.1. (Review the discussion in Section 15.4 about possible side-effects of this instruction.)

15.4.4. The first BCR branches to address zero, where something undesirable is likely to happen; the second BCR "drains the instruction pipeline" and then continues execution.

Section 15.5

15.5.1. The LC values are: (1) X'0246' (2) X'0248' (3) X'0246' (4) X'024A' (5) X'0254' (6) X'0246'

15.5.2. The length and value attributes are:

Sym	Len	Value
A	3	X'000743'
B	4	X'000748'
C	7	X'000743'
D	10	X'00074C'
E	4	X'00074C'
F	1	X'000754'

15.5.3. c(D) = X'C440C4C340C37DC4C37D'.

15.5.4. The values and Length Attributes are:

```
1. Value = X'346', Length = 4
2. Value = X'34A', Length = 4
```

15.5.5. Either zero, two, four, or six bytes of No-Ops, depending on the current value of the Location Counter.

328 Modern processors try to improve the performance of branch instructions by maintaining a "Branch History Table" containing the address of recent branch instructions, whether or not they branched, and their branch addresses. This lets the CPU guess more reliably whether an instruction will or won't branch, and whether it might need to begin pre-fetching instructions at the branch address. Adding branch instructions like `BC 15,*+4` may force other useful entries out of the Branch History Table, possibly slowing execution of your program.

Section 15.6

15.6.1. Every branch condition specified by a set of mask bits has a complementary branch condition determined by forming the ones' complement of the mask bits. Thus, pairs of branches such as

```
         BNZ   Next
         B     LOOP
Next     - - -
```

should be replaced by `BZ  LOOP`.

15.6.2. If we think of the extended mnemonics as representing separate instructions, then this description is not misleading, but it implies that the opcode is 12 bits long. It is more accurate to say that the extended mnemonics provide statements that simplify the many ways to write conditional branch instructions.

15.6.3. An example of an instruction sequence:

```
         LT    0,VAL              Load and test, set CC
         BP    POS                Branch if positive
         BM    NEG                Branch if negative
         B     ZERO               Branch if zero
```

The last instruction is **B**. Should it be **BZ** instead?

15.6.4. The operation code X'40' is the STH instruction, so that the "instruction" X'40404040' appears to be the RX-type instruction

```
         STH   4,X'040'(0,4)      Store a halfword
```

Depending on the contents of GR4 (the apparent base register), its contents could be stored almost anywhere, 64 bytes beyond the address in GR4. Because there are 132 space characters, the instruction could be repeated 33 times if it does not cause a program interruption for a protection or addressing exception. The "instruction" executed after the 33rd STH will depend on what follows the constant named **Target** in memory at execution time.

Section 15.7

15.7.1. The EQU statement is actually a comment statement! If it was in exactly the right place, his "solution" might work for a while, but if instructions between the BZ and the intended target were added or removed, the target location *+18 would be incorrect. This is why using * in branch instructions is a poor programming practice.

Section 15.8

15.8.1. All instructions begin on a halfword, or even, boundary.

Section 15.9

15.9.1. Yes. Many first-generation computers used branch instructions tested the sign bit of an "Accumulator" register. Later machines such as the IBM 7090 provided similar tests, and included comparison instructions that skipped none, one, or two of the following instructions, depending on whether the result of the comparison determined that the Accumulator's contents were less than, equal to, or greater than the memory operand.

Most later machines had status flags such as "Accumulator Overflow" that served some of the function of a condition code.

15.9.2. Since the CC is a 2-bit unsigned integer, only a single value can be assigned in a single instruction execution.

Section 16 Solutions

Section 16.2

16.2.1. The BNM instruction will branch to **ST** if overflow occurs, so the result in c(ANS) = c(X)+c(Y) will undoubtedly be incorrect.

16.2.2. This solution uses "mathematical induction": if (1) you can show that an assumption holds for the number 1, and (2) you can *also* show that if you assume it holds for a positive integer N then the assumption also holds for N+1, *then* (3) your assumption holds for all positive integers.

We assume that the sum of the first N odd integers is N^2.

- Our assumption is true for 1, since $1=1^2$.
- Now we need to show that if the sum of the first N odd integers is N^2, it is true that the sum of the first N+1 integers is $(N+1)^2$.
- Now, the sum of the first N odd integers is

  [1 + 3 + 5 + ... + 2×N-1]

  and the sum of the first N+1 odd integers is

  [1 + 3 + 5 + ... + 2×N-1] + [2×(N+1)-1]

  The last term is $2\times N+2-1=2\times N+1$, so we can write

  [1 + 3 + 5 + ... + 2×N-1] + [2×(N+1)-1] = [N^2] + 2×N + 1

  which is the same as $(N+1)^2$.

So our assumption is true for all positive integers.

16.2.3. The following instructions test for the possibility that N may be 1, so that only a single odd number (1) is stored.

```
         LH    3,NN               Get the value of N from c(NN)
         LM    6,9,=F'0,2,1,1'    Load GR6-GR9 with 0,2,1,1
ADDUP    AR    6,8                Add odd integer to sum in GR6
         SR    3,9                Decrease N by 1
         BZ    ST                 If it was 1, exit the loop
         AR    8,7                Next odd integer in GR8
         B     ADDUP              Branch back (N-1) times
ST       ST    6,SUM              Store result in GR6 at SUM
         - - -
NN       DC    H'6'               Number of odd numbers to add
SUM      DS    F                  Sum of the first c(NN) odd numbers
```

A useful exercise is to write a short program to test both forms of this instruction sequence: this one and the one in Figure 82 on page 218.

16.2.4. The values at **ONE** are alternately 1 and 3; other references to **ONE** probably did strange things! This "technique" is useful when you want to alternate between two values x and y: subtract x from x+y and save the difference for the next subtraction.

16.2.5. There are many ways to count the number of 1-bits in a word. This solution uses a table of bytes, each of which contains a number giving the number of 1-bits in the byte whose value is the offset of that byte from the first byte in the table. For example, the byte at offset 241 = X'F1' = B'11110001' contains 5, the number of 1-bits in B'11110001'.

```
         SR    0,0                Answer value in GR0
         SR    1,1                Work register
Byte1    ICM   1,1,Data           Get 1st byte of Data
         BZ    Byte2              Skip if zero, no bits to count
         IC    1,T(1)             Get the count of 1-bits in this byte
         AR    0,1                Add to answer
Byte2    ICM   1,1,Data+1         Get 2nd byte of Data
         BZ    Byte3              Skip if zero, no bits to count
         IC    1,T(1)             Get the count of 1-bits in this byte
         AR    0,1                Add to answer
Byte3    ICM   1,1,Data+2         Get 3rd byte of Data
         BZ    Byte4              Skip if zero, no bits to count
         IC    1,T(1)             Get the count of 1-bits in this byte
```

```
         AR    0,1                Add to answer
Byte4    ICM   1,1,Data+3         Get 4th and last byte of Data
         BZ    Store              Skip if zero, no bits to count
         IC    1,T(1)             Get the count of 1-bits in this byte
         AR    0,1                Add to answer
Store    STH   0,NBits            Store final bit count
```

This is the table with the "bits-per-byte" counts:

```
T        DC    AL1(0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4)   00-0F
         DC    AL1(1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5)   10-1F
         DC    AL1(1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5)   20-2F
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   30-3F
         DC    AL1(1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5)   40-4F
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   50-5F
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   60-6F
         DC    AL1(3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7)   70-7F
         DC    AL1(1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5)   80-8F
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   90-9F
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   A0-AF
         DC    AL1(3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7)   B0-BF
         DC    AL1(2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6)   C0-CF
         DC    AL1(3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7)   D0-DF
         DC    AL1(3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7)   E0-EF
         DC    AL1(4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8)   F0-FF
```

16.2.6. As in the previous solution, we'll use a table to simplify. Each byte in the table has a number corresponding to the number of leading 0-bits in the byte whose numeric value gives its offset from the start of the table. For example, the byte at offset 26=X'1A'=B'00011010' contains 3, the number of leading 0-bits in B'00011010'.

```
         LA    0,31               Answer in GR0; initialize to 31
         SR    1,1                Work register in GR1
Byte1    ICM   1,1,Data           Get 1st byte of Data
         BP    Finish             Have found first nonzero bit; exit
         BZ    Byte1Z             Skip if zero, test following byte
         LCR   0,0                Negative argument; set error value
         B     Store              And store the result
Byte1Z   SH    0,=H'8'            Subtract 8 from GR0; no 1-bits found
Byte2    ICM   1,1,Data+1         Get 2nd byte of Data
         BNZ   Finish             Have found first nonzero bit; exit
         SH    0,=H'8'            Subtract 8 from GR0; no 1-bits found
Byte3    ICM   1,1,Data+2         Get 3rd byte of Data
         BNZ   Finish             Have found first nonzero bit; exit
         SH    0,=H'8'            Subtract 8 from GR0; no 1-bits found
Byte4    ICM   1,1,Data+3         Get 4rd byte of Data
         BNZ   Finish             Have found first nonzero bit; exit
*        If zero, c(Data) is identically zero, so result will be -1
Finish   IC    1,T(1)             Get number of leading 0-bits in byte
         SR    0,1                Deduct from running count for answer
Store    STH   0,MaxPow           Store the result
*
T        DC    AL1(8,7,6,6,5,5,5,5),8AL1(4)
         DC    16AL1(3),32AL1(2),64AL1(1),128AL1(0)
```

There are other, simpler, and better ways to do this!

16.2.7. Because the first two bytes of the constant at **NN** are zero, the program would have looped until it either ran out of time, or was interrupted for a fixed-point overflow in GR6.

16.2.8. The assembled program looks like this:

```
Loc   Object Code        Assembler Language Statements
                         Ex16_2_8 Start X'5000'
5000  0D40                        BASR  4,0
____                              Using *,4
5002  1B22                        SR    2,2
5004  4320 401D                   IC    2,XX+3
5008  1202                        LTR   0,2
500A  4780 4010                   BZ    Looper
500E  4000 401A                   STH   0,XX
5012  47F0 4010          Looper   B     *         Loop forever here
5016  E4878840           YY       DC    CL4'Ugh'
501A  0000                                        (Skipped bytes!)
501C  FFFFFFF6           XX       DC    F'-10'
```

16.2.9. The contents of the word at **XX** will be X'00F6FFF6'.

16.2.10. Yes. Consider either of these instruction sequences:

```
L     0,=X'7FFFFFFF'          L     1,=X'80000000'
AH    0,=H'1'                 SH    1,=H'1'
```

In both cases the magnitude of the result will be too large to represent correctly in 32 bits.

16.2.11. The first two instructions clear the work registers (R0 and R1) so that the original contents of their low-order bytes can't cause a carry into the high-order 24 bits.

```
SR    0,0               Clear GR0
SR    1,1               Clear GR1
ICM   0,B'1110',X       Get value at X
ICM   1,B'1110',Y       Get value at Y
AR    0,1               Add the values
STCM  0,B'1110',W       Store the result
BO    Over              Branch if the sum overflowed
```

16.2.12. The constant is X'74CBB1'.

16.2.13. The first pair of instructions sets GR1 to zero, and then branches to address zero (not a good idea!). The second pair of sets GR0 to zero and then does not branch; BCR 15,0 is a no-operation instruction.

16.2.14. The assembled program looks like this:

```
Loc   Object Code        Assembler Language Statements
8000                     Ex16_2_E Start X'8000'
8000  0D40                        BASR  4,0
8002                              Using *,4
8002  9812 4016                   LM    1,2,Value
8006  BE27 401E                   STCM  2,B'111',First
800A  1301                        LCR   0,1
800C  47A0 4012                   BC    10,*+8
8010  4001 4022                   STH   0,Last(1)
8014  07FE                        BCR   15,14         Followed by ...
8016  0000                                            Two padding bytes!
8018  00000005           Value    DC    F'4'
801C  FFFFFFFA                    DC    F'-6'
8020                     First    DS    F
8024                     Last     DS    H'-10'
                                  End   Ex16_2_E
```

16.2.15.

1. CC=3, c(GR2) = X'13579BDE'
2. CC=3, c(GR2) = X'80000001'
3. CC=2, c(GR2) = X'00010147' (set by ICM)

16.2.16.

1. CC=0, c(GR1) = X'00000000'
2. CC=3, c(GR2) = X'80000000'
3. CC=1, c(GR3) = X'FDB97530'
4. CC=2, c(GR4) = X'77777777'

Section 16.3

16.3.1. Two possible instruction sequences are:

```
(1)      LH    1,HW          Load and extend 16 bits to 32 in GR1
         AGFR  0,1           Extend 32-bit value to 64 and add

(2)      STG   0,DTemp       Save c(GG0) temporarily
         LGH   0,HW          Load and extend 16 bits to 64
         AG    0,DTemp       Add original contents of GG0
         - - -
DTemp    DS    FD            Temporary storage for c(GG0)
```

Both sequences have disadvantages: (1) requires an additional register, and (2) requires two accesses to a doubleword in memory.

16.3.2. Two possible instruction sequences are:

```
(1)      LH    1,HW          Load and extend 16 bits to 32 in GR1
         SGFR  0,1           Extend 32-bit value to 64 and subtract

(2)      STG   0,DTemp       Save c(GG0) temporarily
         LGH   0,HW          Load and extend 16 bits to 64
         LCGR  0,0           Complement the original halfword value
         AG    0,DTemp       Add original contents of GG0
         - - -
DTemp    DS    FD            Temporary storage for c(GG0)
```

Both sequences have disadvantages: (1) requires an additional register, and (2) requires two memory accesses to a doubleword in memory and two additional instructions. (1) is preferable unless all the other general registers contain data or addresses that you don't want to have to save briefly in memory.

Section 16.4

16.4.1. Because a comparison does not produce an arithmetic result — only a CC setting — no overflow can occur.

16.4.2. If an overflow occurs, the difference must be nonzero; thus the "sense" of the comparison is either "greater-than" or "less-than". If for example you perform the internal arithmetic for comparing 0 to X'80000000', you will see that the CPU should reverse the "sense", as implied by the sign of the internal difference. Practically speaking, if the result of nonzero, the CPU need only invert the sign bit when overflow occurs, and then set the CC in its usual way.

16.4.3. Because the CH instruction compares c(GR1) (which is zero) to only the first two bytes of =F'5', the CC setting will be 0, implying that the operands are equal. (It's easy to make this type of programming error!)

16.4.4. The assembled program looks like this:

```
Loc   Object Code        Assembler Language Statements
                         Ex16_4_4 Start X'4800'
4800  0DA0                        BASR  10,0
4802                              Using *,10
4802  5800 A056          Loop     L     0,One
4806  5A00 A056                   A     0,One
480A  5000 A052                   ST    0,Number
480E   <other ops>                PrintOut Number
4824  5900 A05A                   C     0,Ten
4828  4740 A000                   BL    Loop
482C   <other ops>                PrintOut *
4854  00000000           Number   DC    F'0'
4858  00000001           One      DC    F'1'
485C  0000000A           Ten      DC    F'10'
                                  End   Ex16_4_4
```

Section 16.5

16.5.2. First, we write the original instruction sequences in Figures 90 and 91 in two columns:

```
         L    0,=F'-1'                  LM    0,1,ARG
         LR   1,0                       LCR   0,0
         SL   0,ARG                     LCR   1,1
         SL   1,ARG+4                   BZ    XXX
         AL   1,=F'1'                   SL    0,=F'1'
         BC   B'1100',NoC           XXX STM   0,1,ARG
         AL   0,=F'1'
   NoC   STM  0,1,ARG
```

The only operand that properly causes overflow is the 64-bit maximum negative number X'8000000000000000'. The instructions in the first column use only logical addition and subtraction, so they cannot cause an overflow condition. Both LCR instructions in the second column can potentially cause an overflow condition, so we would have to take steps to suppress a Fixed-Point Overflow interruption. If we do this, then both instruction sequences could be followed by these:

```
         LTR   1,1            Check low-order word for zero
         BNZ   NoOver         If it's nonzero, operand can't overflow
         C     0,=A(X'80000000')  Check high-order word for max neg #
         BE    Overflow       64-bit complementation would indicate overflow
NoOver   - - -
```

16.5.3. Consider the following instructions:

```
         LM    0,1,A          Get all 64 bits of 1st operand
         AL    1,B+4          Add low-order half of second operand
         BC    12,NoCarry     Branch if no carry, it's easy
CProp    A     0,=F'1'        Now add the carry
         BNO   NoCarry        If no overflow, finish up normally
Carry    A     0,B            Now add high-order halves
         BNO   Overflow       If no oflo now, sum has overflowed
         B     Okay           Sum is correct
NoCarry  A     0,B            Add high-order halves normally
Okay     STM   0,1,C          64-bit sum stored at C
         BO    Overflow       If overflow now, it's true
```

If an overflow occurs in propagating the low-order carry at the instruction named **CProp**, the original contents of GR0 (the first word of the first operand) must have been X'7FFFFFFF'. Thus, when adding the high-order words at the instruction named **Carry**, if *no* overflow occurs, the contents of the fullword at **B** must have been nonnegative, implying that the 64-bit sum must have overflowed.

This solution isn't obvious; try writing a program with some sample data to test it.

16.5.5. With the LCR instructions, there's no easy way to avoid the possibility of intermediate overflow conditions; and there is no “logical complement” instruction to help. Thus, the sequence shown in Figure 90 on page 225 could be used.

We'll see in Section 16.9 the instructions that let us modify the bits in the Program Mask.

16.5.11. This figure shows how the CC bits are set:

CC bits	Second bit=0	Second bit=1
First bit=0	zero, no carry	nonzero, no carry
First bit=1	zero, carry	nonzero, carry

16.5.12. Subtraction requires adding the ones' complement of the second operand (which will be all 1-bits) and a low-order 1-bit to the first operand (which will force a carry along the complete length of the sum, and off the left end). The CC value B'10' reflects the zero result, with a high-order carry.

Section 16.8

16.8.1. If the CC settings following SL or SLR are 1, 2, or 3, then the settings following CL or CLR (with the same operands) will be 1, 0, or 2 respectively. Remember that the CC cannot be 0 following SL or SLR; see Exercise 16.4.1.

16.8.2. This is most easily done (now) with a CLM and CLMH instruction.

```
         CLMH  0,B'0011',StgOp    Compare bytes 2 and 3 of GG0
         BL    RegLow             c(GG0) is low
         BH    RegHigh            c(GG0) is high
         CLM   0,B'1100',StgOp+2  Compare bytes 4/5 of GG0 (0/1 of GR0)
         BL    RegLow             c(GG0) is low
         BH    RegHigh            c(GG0) is high
         BE    RegEqual           c(GG0) = selected bytes of StgOp
```

If a temporary register is available, the shift instructions we'll discuss in Section 17 can solve this problem more easily.

16.8.3. After executing the two instructions with the first pair of operands, the CC setting is 1 (c(GR0) < c(GR1)) for CR and 2 (c(GR0) > c(GR1)) for CLR. After executing the two instructions with the second pair of operands, the CC setting is 2 (c(GR0) > c(GR1)) for CR and 1 (c(GR0) < c(GR1)) for CLR.

This technique is often used in sorting, where many other types of data are compared logically. By inverting the sign bit of binary integers, (formerly) negative numbers are now logically less than (formerly) positive numbers. After the comparisons are complete, the sign bits are inverted again to restore the original values of the numbers.

16.8.5. With arithmetic comparison, all non-negative values are greater than all negative values; with logical values, all "negative" values are greater than all non-negative values. For example, compare +1 = X'00000001' and −1 = X'FFFFFFFF'.

16.8.6. Consider this test program:

```
X16_8_6  CSect ,
         Using *,15
         L     0,Oldest
         L     1,Later
         L     2,Newest
         LR    3,1                Copy Later value to GR3
         SLR   3,0                Subtract Oldest value
         LTR   3,3                Test result
         Printout 3
         LR    3,2                Copy Newest value to GR3
         SLR   3,1                Subtract Later value
         LTR   3,3                Test result
         Printout 3,*,Header=No
*
         DS    0F                 Align on fullword
Oldest   DC    X'FFFFFFFE'        Oldest value
Later    DC    X'FFFFFFFF'        A later value
Newest   DC    X'00000001'        Most recent value
         EndX16_8_6
```

The calculated values in GR3 are both positive, as expected.

16.9.3.

```
         BALR  1,0                Program mask in GR1
         SLL   1,4                Drop off ILC and CC bits
         ICM   0,B'1000',CCode    New CC value in GR0
         SRDL  0,4                Shift into GR1
         SPM   1                  Set new CC without changing pm
```

16.9.4.

```
         BALR  0,0                ILC, CC, PM, and IA in GR0
         SLL   0,2                Drop ILC bits
         SRL   0,30               Shift to right end of GR0
         ST    0,CCode            Store Condition Code
```

16.9.5.

```
         BALR  0,0                ILC, CC, PM, and IA in GR0
         STCM  0,B'1000',PMask    Store that leftmost byte
         NI    PMask,B''1111'     Set all but Program Mask to zeros
```

Programming Problem 16.1.

The key to solving this problem is to define all the terms in A-type address constants.

```
*  Display 3 arithmetic expressions as a 12-byte character string.
*  c(X) = B'100000000000000' + X'C7A98' - 231471192,
*  c(Y) = X'C0FFEE' - C'@#$' - 694895668, and
*  c(Z) = 1073741824 + X'F194F6' + X'ABCD'.
P16_1  Start 0
       Print Nogen
       Using *,15
       B     Start
*
X1     DC    A(B'100000000000000')
X2     DC    A(X'C7A98')
X3     DC    A(231471192)
*
Y1     DC    A(X'C0FFEE')
Y2     DC    A(C'@#$')
Y3     DC    A(694895668)
*
Z1     DC    A(1073741823)
Z2     DC    A(X'F194F6')
Z3     DC    A(X'ABCD')
*
XANS   DS    0XL12
ANS    DS    0CL12
X      DS    F
Y      DS    F
Z      DS    F
*
Start  DC    0H
       L     0,X1
       A     0,X2
       S     0,X3
       ST    0,X
*
       L     1,Y1
       S     1,Y2
       S     1,Y3
       ST    1,Y
*
       L     2,Z1
       A     2,Z2
       A     2,Z3
       ST    2,Z
*
       Printout XANS,ANS,*
       End
```

Your printed output will look something like this:

```
 XANS     = X'F240C240D6D9405F40F240C2'
 ANS      = '2 B OR ¬ 2 B'
```

Programming Problem 16.2.

The key to solving this problem is to define all the terms with length attribute 4. This is different from Problem 16.1, because the alignment and padding of the terms is different.

```
*  Display 4 arithmetic expressions as a 16-byte character string.
*   c(W) = c(WA) + c(WB) - 929065920, where
*            c(WA) = B'100000000000000',
*            c(WB) = X'1230000'.
*   c(X) = c(XA) + 50344169 + c(XB), where
*            c(XA) = X'5CF17',
*            c(XB) = C'000'.
*   c(Y) = c(YA) + c(YB) + c(YC), where
*            c(YA) = B'11111111',
*            c(YB) = X'1261F02',
*            c(YC) = C'ABCD'.
*   c(Z) = c(ZA) + c(ZB) - c(ZC), where
*            c(ZA) = X'CAF75A',
*            c(ZB) = B'1000011',
*            c(ZC) = 511686493.
P16_2  Start 0
       Print Nogen
       Using *,15
       B     Start
*
CANS   DS    0CL16
XANS   DS    0XL16
W      DS    F
X      DS    F
Y      DS    F
Z      DS    F
Start  DC    0H
       L     0,=BL4'100000000000000'
       A     0,=XL4'1230000'
       S     0,=F'929065920'
       ST    0,W
*
       L     1,=XL4'5CF17'
       A     1,=F'50344169'
       A     1,=CL4'000'
       ST    1,X
*
       L     2,=BL4'11111111'
       A     2,=XL4'1261F02'
       A     2,=C'ABCD'
       ST    2,Y
*
       L     3,=XL4'CAF75A'
       A     3,=BL4'1000011'
       S     3,=F'511686493'
       ST    3,Z
       Printout CANS,XANS,*
       End
```

Your printed output will look something like this:

```
 CANS     = 'IBM 360 BYTES.  '
 XANS     = X'C9C2D440F3F6F040C2E8E3C5E24B4040'
```

Programming Problem 16.3.

```
*   c(W) = c(WA) + c(WB) - 759375551, where
*            c(WA) = B'100000000000000',
*            c(WB) = X'CBA98'.
*   c(X) = c(XA) - c(XB) + 1386388536, where
*            c(XA) = X'COFFEE',
*            c(XB) = C'@#$'.
*   c(Y) = c(YA) + c(YB) + c(YC), where
*            c(YA) = B'11111111',
*            c(YB) = X'1F7C05',
*            c(YC) = C'ABCD'.
*   c(Z) = c(ZA) + c(ZB) - 975583924, where
*            c(ZA) = X'FFFF',
*            c(ZB) = -65536.
P16_3  Start 0
       Print Nogen
       Using *,15
       B     Start
*
CANS   DS    0CL16
XANS   DS    0XL16
W      DS    F
X      DS    F
Y      DS    F
Z      DS    F
*
Start  DC    0H
       L     0,=BL4'100000000000000'
       A     0,=XL4'CBA98'
       S     0,=F'759375551'
       ST    0,W
*
       L     1,=XL4'COFFEE'
       S     1,=CL4'@#$'
       A     1,=F'1386388536'
       ST    1,X
*
       L     2,=BL4'11111111'
       A     2,=XL4'1F7C05'
       A     2,=C'ABCD'
       ST    2,Y
*
       L     3,=XL4'FFFF'
       A     3,=A(-65536)
       S     3,=F'975583924'
       ST    3,Z
*
       Printout CANS,XANS,*
       End
```

Your printed output will look something like this:

```
 CANS     = 'KILROY WAS HERE.'
 XANS     = X'D2C9D3D9D6E840E6C1E240C8C5D9C54B'
```

Programming Problem 16.4.

The three values starting at **Consts** define initial values that make it easier to start the sequence.

```
        Title 'Solution to Problem 16.4'
P16_4   START 0
        BASR  15,0                Execution-time base register
        USING *,15                Establish addressability
        LM    0,2,Consts          Initialize first 3 terms
        L     3,Count             Set print counter
LOOP    LR    4,0                 Move oldest term
        AR    4,1                 Add middle term
        AR    4,2                 And youngest term
        ST    4,Value             Store for printing
        PRINTOUT Value            Print it next
        LR    0,1                 Move middle to oldest
        LR    1,2                 And previous new to middle
        LR    2,4                 And new one to youngest
        S     3,One               Subtract 1 from term counter
        BP    LOOP                Branch if still positive
        PrintOut *                Stop here
Consts  DC    F'-1,0'             Previous 3 terms of sequence, with..
One     DC    F'+1'               Constant +1, third starting term
Count   DC    F'25'               Number of terms
Value   DS    F                   Space for printed value
        END   P16_4
```

This solution requires that the constant named **One** immediately follow the two constants named **Consts**. This is *not* a good programming practice; suppose someone replaced references to **One** with the literal =F'1'?

The 25th term is X'0011CB8C' = 1166220.

This solution uses an array of values, adding three and storing the sum at the fourth. Halfword values can't be used because the value of the 21st term exceeds 15 bits.

```
        Title 'Solution to Problem 16.4'
P16_4A  START 0
        BASR  15,0                Execution-time base register
        USING *,15                Establish addressability
        LH    0,=H'22'            Count of new terms
        LM    2,4,Table           Get first three values
        PrintOut 2,3,4,Header=No Print the first 3 terms
        SR    1,1                 Initialize index
Loop    L     2,Table(1)          Get oldest value
        A     2,Table+4(1)        Add second-oldest value
        A     2,Table+8(1)        Add most recent value
        ST    2,Table+12(1)       Store in new position in table
        PrintOut 2,Header=No      Print the new value
        AH    1,=Y(L'Table)       Increment the index
        SH    0,=H'1'             Decrement count
        BP    Loop                Repeat until count = 0
        PrintOut *,Header=No      Stop
        LtOrg ,                   Place literals here
Table   DC    F'0,1,2'            Three initial values
        DS    22F                 Remaining values
        END   P16_4A
```

Programming Problem 16.7.

```
P16_7    Start 0
         Using P16_7,15         Provide addressability
* Set up initial values.
         LA    2,0              Previous value.
         LA    3,1              Current value.
Loop     PrintOut 3             Print a value
         LR    4,2              Get a copy of the previous value.
         AR    4,3              Compute the next value
         C     4,=F'1000000'    Compare to one million
         BNL   Done             If sum overflowed, we're done
         LR    2,3              Copy current value to previous.
         LR    3,4              Copy next value to current.
         B     Loop             Repeat the calculation
Done     PrintOut *,Header=No   Terminate the program
         End   P16_7
```

The output will look something like this:

```
 GPR   3  = X'00000001' =          1
 GPR   3  = X'00000001' =          1
 GPR   3  = X'00000002' =          2
 . . . . . . . . . . . . . . . . . .
 GPR   3  = X'00000003' =          3
 GPR   3  = X'0004D973' =     317811
 GPR   3  = X'0007D8B5' =     514229
 GPR   3  = X'000CB228' =     832040
```

You may want to revise your solution to use the CONVERTO macro instead of PRINTOUT. (See Programming Problem 16.9.)

Programming Problem 16.8.

```
P16_8    Start 0
         Using P16_8,15         Provide addressability
         LA    2,0              Previous value.
         LA    3,1              Current value.
Loop     PrintOut 3             Print a value
         LR    4,2              Get a copy of the previous value.
         AR    4,3              Compute the next value
         BO    Done             If sum overflowed, we're done
         LR    2,3              Copy current value to previous.
         LR    3,4              Copy next value to current.
         B     Loop             Repeat the calculation
Done     PrintOut *,Header=No   Terminate the program
         End   P16_8
```

The printed output looks something like this:

```
 GPR   3  = X'00000001' =          1
 GPR   3  = X'00000001' =          1
 GPR   3  = X'00000002' =          2
 GPR   3  = X'00000003' =          3
 . . . . . . . . . . . . . . . . . .
 GPR   3  = X'43A53F82' = 1134903170
 GPR   3  = X'6D73E55F' = 1836311903
```

It's not always a good idea to test for overflow, as an interruption may have occurred. Here are three other ways to test for the largest value:

1. Check for the largest number:

   ```
   C     r,=F'1836311903'   r = register with current number
   BE    Done
   ```

   - Not an “honest” method, because you have to know the last value!

2. Use a logical addition that doesn't cause overflow, and check for an arithmetically negative result:

```
         ALR   r,rprev            rprev = register with previous number
         LTR   r,r                Test sign of r
         BNP   DONE               If not +, previous number was last
```

3. After output, test the current value against maximum positive number:

```
         L     t,=F'2147483647'   Maximum positive integer in temp reg
         SR    t,r                Subtract current from max
         CR    t,rprev            Compare difference to previous value
         BL    DONE               If smaller, current number is last
```

- If (max-current) < (previous), we know that previous+current will overflow.

Programming Problem 16.9.

Note that the DC statement labeled `Title` starts in column 15, so that the constant need not be continued (the terminal apostrophe is in column 71).

```
P16_9    Start 0
         Using P16_9,15         Provide addressability
* Set up initial values.
         LA    2,0              Previous value.
         LA    3,1              Current value.
         PrintLin Title,TLen    Print a header line.
         PrintLin OutRec,1      Print a blank line.
Loop     CONVERTO 3,OutRec      Convert the number to characters
         PrintLin OutRec,OutLen Print the results.
         LR    4,2              Get a copy of the previous value.
         AR    4,3              Compute the next value
         BO    Done             If sum overflowed, we're done
         LR    2,3              Copy current value to previous.
         LR    3,4              Copy next value to current.
         B     Loop             Repeat the calculation
Done     PrintLin EndRec,EndLen Print an ending line
         PrintOut *,Header=No   Terminate the program
OutRec   DC    CL12' '          Output from CONVERTO
OutLen   Equ   *-OutRec
Title    DC   C'1Fibonacci Sequence to Maximum Positive Fullword Value'
TLen     Equ   *-Title
EndRec   DC    C'0Program ends.'
EndLen   Equ   *-EndRec
         End   P16_9
```

The output lines will look like this; note that the carriage control characters are shown.

```
1Fibonacci Sequence to Maximum Positive Fullword Value

           1
           1
           2
           3
 . . . . . .
   701408733
  1134903170
  1836311903
0Program ends.
```

Programming Problem 16.10.

```
P16_10  Start 0
        USING P16_10,15         Establish addressability
        SGR   2,2               Previous value in GG2; F(0)=0.
        LGB   3,Byte1           Current value in GG3; F(1)=1.
        PrintLin Title,TLen     Print a title line
        PrintLin OutRec,1       Print a blank line
Loop    CONVERTO 19,OutRec      Convert integer in GG3 to characters
        PrintLin OutRec,OutLen  Print the results.
        LGR   4,2               Get a copy of the previous value.
        AGR   4,3               Compute the next value.
        BO    Done              Overflow means we're done.
        LGR   2,3               Copy current value to previous.
        LGR   3,4               Copy next value to current.
        B     Loop              Repeat the loop
Done    PrintLin EndRec,EndLen  Print an ending line.
        PrintOut *,Header=No    Terminate the program
Byte1   DC    FL1'1'            Initial value for F(1)
OutRec  DC    CL21' '           Carriage control for output.
OutLen  Equ   *-OutRec          Length of record.
Title   DC    C'1Fibonacci Sequence to Maximum Doubleword Value'
TLen    Equ   *-Title
EndRec  DC    C'0Program ends.'
EndLen  Equ   *-EndRec
        End   P16_10
```

The output looks like this (with carriage control characters):

```
1Fibonacci Sequence to Maximum Doubleword Value

                    1
                    1
                    2
  . . . . . . . . . .
  2880067194370816120
  4660046610375530309
  7540113804746346429
0Program ends.
```

Programming Problem 16.11.

The assembled program looks like this:

```
000000                00000 0001A    1 P16_11  CSect ,
                 R:C  00000          2         Using *,12
000000 18CF                          3         LR    12,15
000002 5AF0 C008            00008    4         A     15,X
000006 0DCF                          5         BASR  12,15
000008 00000012                      6 X       DC    F'18'
00000C 00000004                      7         DC    F'4'
000010 07FE                          8 Exit    BR    14
000012 58A0 C004            00004    9         L     10,X-4
000016 47FA C004            00004   10         B     X-4(10)
000000                              11         End   P16_11
```

The key to following its execution is to *not* study the source code, because the BASR instruction changes the contents of GR12, the base register! You must calculate effective addresses carefully at each step.

Programming Problem 16.12.

Your results might look like these:

```
*** PRINTOUT requested at Address 01A008, Statement     7, CC=0
GPR   1  = X'00000000' =          0
*** PRINTOUT requested at Address 01A040, Statement   758, CC=3
GPR   2  = X'80000000' = -2147483648
*** PRINTOUT requested at Address 01A078, Statement   776, CC=1
GPR   3  = X'FDB97530' =   -38177488
*** PRINTOUT requested at Address 01A0B0, Statement   794, CC=2
GPR   4  = X'77777777' =  2004318071
*** PRINTOUT requested at Address 01A0E0, Statement   809, CC=2
```

Section 17 Solutions

Section 17.1

17.1.1. (1) 7, (2) 56, (3) 33, (4) 45, (5) 63, (6) 1, (7) 0.

Section 17.2

17.2.1. To find the bit, its byte address is simply K divided by 8, and the remainder of the division gives the position of the bit within that byte.

```
         L     2,KK              Get bit number K
         SRDL  2,3               Byte count in GR2
         SRL   3,29              And bit number in GR3
         IC    1,BStrg(2)        Get desired byte in GR1
         SLL   1,24              Position at left end of GR1
         SLL   1,0(3)            Shift off undesired bits
         SR    0,0               Now clear GR0
         SLDL  0,1               And move the bit into GR0 from GR1
```

17.2.2. First, we use a right shift:

```
         L     1,K               Shift amount in GR1
         L     0,=A(X'80')       Put a 1-bit in byte-position zero
         SRL   0,0(1)            Shift correct number of places
         STC   0,KthBit          Store result bit
```

Another solution, shifting left:

```
         L     1,=F'7'
         S     1,K               Compute 7-K
         L     0,=F'1'           1-bit in GR0, byte-position 7
         SLL   0,0(1)            Shift left 7-K places
         STC   0,KthBit          Store the k-th bit
```

17.2.3. The table at **BTbl** contains 8 bytes with a 1-bit in a position corresponding to its offset from **BTbl**.

```
         L     1,K               Load bit counter into GR1
         IC    1,BTbl(1)         Insert byte with indexing
         STC   1,KthBit          Store result
         - - -
BTbl     DC    X'8040201008040201'   Bytes with bits 0-7
```

17.2.4. Even though the register is only 32 bits long, the shift amount is determined from the *six* (not five!) low-order bits of the Effective Address. The shift will set all bits in GR0 to zero.

17.2.5. If you want to use the shift instructions, you could write:

```
         IC    1,DPG+3           c(GR1)=xxxxxx78
         SLL   1,8               c(GR1)=xxxx7800
         IC    1,DPG+2           c(GR1)=xxxx7856
         SLL   1,8               c(GR1)=xx785600
         IC    1,DPG+1           c(GR1)=xx786534
         SLL   1,8               c(GR1)=78563400
         IC    1,DPG             c(GR1)=78563412
```

With ICM instructions, you could write:

```
         ICM   1,B'0001',DPG+0    c(GR1)=xxxxxx12
         ICM   1,B'0010',DPG+1    c(GR1)=xxxx3412
         ICM   1,B'0100',DPG+2    c(GR1)=xx563412
         ICM   1,B'1000',DPG+3    c(GR1)=78563412
```

17.2.6. These instructions do the job:

```
         SR    1,1                Set GR1 to zero
Test     C     0,=A(X'40000000')  Test value in GR0
         BNL   Done               If c(GR0) not low, we're done
         A     1,=F'1'            Increment shift count
         SLL   0,1                Shift left 1 bit
         B     Test               Repeat test
Done     - - -                    Shift count in GR1
```

17.2.7. The representations of the data at X and Y are c(X)=X'12D687' and c(Y)=X'74CBB1'. These instructions will do the job:

```
         ICM   0,B'0111',X        Put X in right end of GR0
         ICM   1,B'0111',Y        Put Y in right end of GR1
         SLDL  0,8                Shift both left 8 bits
         AR    0,1                Add the values arithmetically
         STCM  0,B'1110',W        Store result
         JO    OverFlo            Branch if the sum overflowed
```

The representation of c(W)=X'87A238'; the sum overflows.

17.2.8. If n=0, nothing happens; if n≠0, calculate n+c(GRn) (modulo 64: the rightmost 6 bits) and shift GRn that amount.

17.2.9. Yes: for signed integers, the LH instruction propagates the sign bit; but if the number is unsigned *and* its leftmost bit is 1, LH treats that bit as a sign that is propagated when it should not be. (Details like this can be very important!)

Section 17.3

17.3.1. Simulating SLDL in this case is fairly easy:

```
         LH    1,NShifts          Get shift amount N
         L     0,DataWord         Load the data word
         SLL   0,0(1)             Shift left N bits
         ST    0,DWord            Store high-order 32 bits
         SR    0,0                Set GR0 to zero
         ST    0,DWord+4          Store low-order 32 bits
```

17.3.2. Because we have assumed that the low-order word is zero, the solution to Exercise 17.3.1 will also work. (Remember that if N>32, all bits in GR0 will be lost, and c(DWord) will be zero.)

17.3.3. In this case, we must handle the portion of the high-order word that shifts into the low-order word of the double-length result.

```
         LH    1,NShifts          Load shift count N into GR1
         L     0,DataWord         Get high-order word into GR0
         SRL   0,0(1)             Shift off low-order bits
         ST    0,DWord            Store high-order half of result
         L     0,DataWord         Reload a copy of original data
         LCR   1,1                Complement shift count, now -N
         SLL   0,32(1)            Drop off high-order (N-32) bits
         ST    0,DWord+4          Store low-order half of result
```

17.3.4. In this case, we check for N≥32: if true, the high-order word of the resulting double-length logical right shift will be zero.

```
         LH    1,NShifts          Load shift count N into GR1
         L     0,DataWord         Get high-order word into GR0
         CH    1,=H'32'           Check shift count
         BL    Under32            Branch if less than 32 shifts
         SRL   0,32(1)            Shift N-32 bits (see note below)
         ST    0,DWord+4          Store low-order half of result
         SR    0,0                Set GR0 to zero
         ST    0,DWord            Store high-order half of result
         B     Done               ... and finish
Under32  SRL   0,0(1)             Shift off low-order bits
         ST    0,DWord            Store high-order half of result
         L     0,DataWord         Reload a copy of original data
         LCR   1,1                Complement shift count, now -N
         SLL   0,32(1)            Drop off high-order (N-32) bits
```

```
         ST    0,DWord+4          Store low-order half of result
Done     - - -
```

The shift amount of the instruction SRL 0,32(1) is 32+N (mod 64); but since N≥32, the result gives the same shift count as N-32.

17.3.5. In this case, the initial data is 64 bits long, and is to be shifted left 0 to 31 bits. Some comment statements have been inserted to clarify the operation of the instructions.

```
         LH    1,NShifts          Get shift count N
         LM    2,3,DWData         64-bit shifting data in (GR2,GR3)
* GR2/GR3:  HHHHH....HHHHHLLLLL....LLLLL  High and low order bits
         LR    0,3                Copy low-order word to GR0
         SLL   0,0(1)             Shift N bits left
* GR0:      LLLL0000...000
         ST    0,DWord+4          Done with low-order half of result
         LR    0,2                Copy high-order word to GR0
         SLL   0,0(1)             Shift left; now done with HO data
* GR0:      HHH...HHH...000
         LCR   2,1                Put -N in GR2
         SRL   3,32(2)            Shift low-order data right 32-N bits
* GR3:      000...000...LLL
         ALR   0,3                Merge the two pieces in GR0
* GR0:      HHH...HHHLLLLLL
         ST    0,DWord            Store high-order half of result
* Result:   HHH...HHHLLLLL LLLL0000...000
```

17.3.6. This solution is similar to that of Exercise 17.3.5:

```
         LH    1,NShifts          Get shift count N
         LM    2,3,DWData         Load 64 bits to be shifted right
         LR    0,2                Copy high-order half to GR0
         SRL   0,0(1)             Shift right N bits
         ST    0,DWord            Store high-order half of result
         LR    0,3                Copy low-order half to GR0
         SRL   0,0(1)             Shift; low bits of low result word
         LCR   3,1                -N in GR3
         SLL   2,32(3)            Shift high-order half 32-N bits
         ALR   2,0                Merge pieces of high/low halves
         ST    2,DWord+4          Store low-order half of result
```

17.3.9. The following sequence makes no checks for whether or not significant bits are lost in the rearrangement. The word containing four integers in the old format is at **OLD**, and the four in their new format are to be stored at **NEW**.

```
         L     0,OLD              Fetch the word with 4 integers
         SRDL  0,6                Shift 6 bits of 4th integer into GR1
         SRL   1,2                Extend to 8 bits
         SRDL  0,13               All 13 bits of 3rd integer into GR1
         SRL   1,2                Extend to 15 bits
         SRDL  0,2                Shift 2 low-order 2nd integer bits
         SRL   0,2                Drop 2 high-order bits of 2nd
         SRDL  0,7                Shift last 7 bits into GR1
         ST    1,NEW              Store final result
```

17.3.10. This first solution works so long as the number of bits allowed for the value of each integer is sufficient; the integers may then be in either the logical or the arithmetic representation.

```
         L     0,Fourth           Get 4th integer
         SRDL  0,6                Shift into GR1
         L     0,Third            Get 3rd integer
         SRDL  0,13               shift into GR1 next to 4th
         L     0,Second           Get 2nd integer
         SRDL  0,4                Shift into GR1 next to 3rd and 4th
         L     0,First            Get 1st integer
         SRDL  0,9                Fill up the new word
         ST    1,NEW              And store the result
```

A second solution using A instructions (to Add) also does the job economically, but *only* if the integers are unsigned. (Explain why this is so.)

```
         L     0,First            Get 1st integer
         SLL   0,4                Make room for second
         A     0,Second           Add 2nd integer
         SLL   0,13               Make room for third
         A     0,Third            Add 3rd integer
         SLL   0,6                Make room for last
         A     0,Fourth           Add 4th and last integer
         ST    0,NEW              Store result
```

17.3.11. This solution is almost identical to that for Exercise 17.3.10.

```
         L     0,Fourth           Get 4th integer
         LH    2,L4               Get bit length for 4th integer
         SRDL  0,0(2)             Shift by specified amount
         L     0,Third            Get 3rd integer
         LH    2,L3               Etc...
         SRDL  0,0(2)             Etc...
```

17.3.12. The solution to Exercise 17.3.11 also works for integers in the arithmetic representation, provided that enough bits are allotted to correctly represent the given values.

17.3.13. In Figure 113 on page 248 the value zero would be stored. In Figure 114 on page 249 the program would stay in an unending loop comprising the SRDL, LTR, and BNM instructions until the program was halted.

17.3.14. For nonnegative values of N, a solution might be:

```
         L     1,NN               Get N
         LR    2,1                Move a copy to GR2
         A     2,=F'1'            Force a carry if low bit = 1
         SRL   2,1                Now have Ceiling(N/2) in GR2
         AR    1,2                Ceiling(N+N/2) now in GR1
```

17.3.15. To set the flag bytes correctly, we must test each integer as it is packed into the result word.

```
         LM    2,3,=F'0,1'        For setting data-fit flags
         L     0,Fourth           Get 4th integer
         SRDL  0,9                Shift 9 bits into GR1
         STC   2,FLAG4            Set 4th-integer FLAG4 to 'fit' value
         LTR   0,0                Now test if it really did fit
         BZ    OKFour             Skip one instruction if it did
         STC   3,FLAG4            Set data-fit FLAG4 to 'no-fit' value
OKFour   L     1,Third            Fetch 3rd integer
         SRDL  0,13               Shift 13 bits into GR1
         STC   2,FLAG3            Set 4th-integer FLAG3 to 'fit' value
         LTR   0,0                Now test if it really did fit
         BZ    OKThree            Skip one instruction if it did
         STC   3,FLAG3            Set data-fit FLAG3 to 'no-fit' value
OKThree  L     0,Second           Get 2nd integer
         SRDL  0,4                Shift 4 bits into GR1
         - - -                    ...similarly for the other integers
```

17.3.16. We will illustrate two solutions: the first moves the bits from GR0 to GR1, then to GR3 and to GR2. (A sketch may help.)

```
         L     4,=F'32'           Set bit counter in GR4
Loop     SRDL  0,1                Shift a bit off right end of GR0
         LR    3,1                Move the shifted bit to GR3
         SLDL  2,1                Move into GR2, going left
         S     4,=F'1'            Count down by 1
         BP    Loop               If count is positive, repeat
         LR    1,2                Leave result in GR1
```

The second solution uses logical addition to test for 1-bits at the left end of R0. (Again, a sketch may help.)

```
         L     4,=F'32'           Set bit counter in GR4
Loop     SRL   1,1                Make room in GR1 for a bit
         ALR   0,0                Force a bit off left end of GR0
         BC    12,Count           If no carry out, it was a zero bit
         AL    1,SignBit          If carry, put a 1-bit in sign of GR1
Count    S     4,=F'1'            Count down by 1
         BP    Loop               Repeat if count is positive
         - - -
SignBit  DC    A(X'80000000')     Leftmost 1-bit in a fullword
```

17.3.19. It works correctly. We'll see in Chapter VI that the LA instruction can replace the literals, and in Chapter VII that the UNPK and TR instructions let us do this type of conversion more easily.

Section 17.4

17.4.1. The final CC setting in this instruction sequence is a correct emulation of SLDA:

```
         SR    0,0                Set GR0 to zero
         ST    0,DWord+4          Store low-order half of result
         LH    1,NShifts          Get shift count
         L     0,DataWord         Get datum to be shifted
         SLA   0,0(1)             Shift left N places
         ST    0,DWord            Store high-order half of result
```

17.4.2. The instruction sequence shown for the solution to Exercise 17.4.1 above can still be used, because the low-order half of the source operand was assumed to be zero.

17.4.3. In this instruction sequence, the final CC setting will not be correct:

```
         LH    1,NShifts          Get shift count N
         L     0,DataWord         Get data to be shifted
         LR    2,0                Copy to GR2 (for low-order bits shift)
         LCR   3,1                Complement N in GR3
         SLL   2,32(3)            Shift 32-N bits left
         ST    2,DWord+4          Store low-order half of result
         SRA   0,0(1)             Now, shift high-order half
         ST    0,DWord            Store high-order half of result
```

The reason the CC setting is not reliable is that if the original operand was nonnegative, a nonzero bit may have been shifted into the low-order half of the result, leaving the high-order half zero. The SLA instruction would indicate CC=0, whereas SRDA would indicate CC=2 for a positive nonzero result.

17.4.4. Here, we must test for 32 or more shifts:

```
         L     0,DataWord         Get data to be shifted
         LR    2,0                Copy it to GR2
         LH    1,NShifts          Get shift count N
         CH    1,=H'32'           Check its size
         BL    Under32            Branch if less than 32
         SRA   2,32(1)            Shift right N-32 bit positions
         ST    2,DWord+4          Store low-order half of result
         SRA   0,32               Shift GR0 32 bits: only sign bits left
         ST    0,DWord            Store high-order half of result
         B     Done               ...and exit
Under32  LCR   3,1                Put -N in GR3
         SLL   2,32(3)            Shift data 32-N places left
         ST    2,DWord+4          Store low-order half of result
         SRA   0,0(1)             Shift high-order half N bit positions
         ST    0,DWord            Store high-order half of result
Done     - - -
```

The CC setting at **Done** will not be correct. The first SRA instruction shifts N-32 positions because N≥ 32, and adding 32 is equivalent to subtracting 32 in the rightmost 6 bit positions of the Effective Address.

17.4.5. In this case, we must shift a 64-bit signed operand:

```
         LH    1,NShifts          Get shift count N
         LM    2,3,DWData         Put 64-bit operand in (GR2,GR3)
         LR    0,3                Copy low-order word to GR0
         SLL   0,0(1)             Shift left N bits
         ST    0,DWord+4          Store low-order half of result
         LR    0,2                Copy high-order word to GR0
         SLA   0,0(1)             Shift left N bit positions
         ST    0,DWord            Store temporarily
         LCR   1,1                Complement to get -N in GR1
         SRL   3,32(1)            Shift low-order word right N-32 bits
         AL    3,DWord            Add in the high-order bits
         ST    3,DWord            Store high-order half of result
```

The CC setting is not correct, because its value will depend on the result of the AL instruction.

17.4.6. We must right-shift a 32-bit operand right N bits:

```
         LH    1,NShifts          Get shift count
         LM    2,3,DWData         Put shift operand into (GR2,GR3)
         LR    0,3                Copy low-order half to GR0
         SRL   0,0(1)             Shift right N bits
         ST    0,DWord+4          Store low-order part temporarily
         LR    0,2                Copy high-order half to GR0
         SRA   0,0(1)             Shift right arithmetically N bits
         ST    0,DWord            Store high-order half of result
         LCR   1,1                Complement N
         SLL   2,32(1)            Shift out unneeded bits of high half
         AL    2,DWord+4          Add back the low-order half's bits
         ST    2,DWord+4          Store low-order half of result
```

The CC setting is not correct.

17.4.10. Because two bits are to be removed from the high-order part of the first and second integers, sign extension is required only for the third and fourth integers.

```
         L     0,OLD              Fetch the word with 4 integers
         SRDL  0,6                Shift 6 bits of 4th integer into GR1
         SRA   1,2                Sign-extend to 8 bits
         SRDL  0,13               All 13 bits of 3rd integer into GR1
         SRA   1,2                Sign-extend to 15 bits
         SRDL  0,2                Shift 2 low-order bits of integer 2
         SRL   0,2                Drop 2 high-order bits of integer 2
         SRDL  0,7                Shift last 7 bits into GR1
         ST    1,NEW              Store final result
```

17.4.11. Since c(NUM) is positive and nonzero, we will count the number of arithmetic right shifts:

```
         L     1,NUM              Get number to be checked
         L     0,=F'1'            Place starting exponent in GR0
Test     SRA   1,1                Shift number right once
         BZ    Done               Finished if now zero
         SLL   0,1                Double the power of 2 in GR0
         B     Test               And try again
DONE     - - -                    2**N now in GR0
```

Check your solution carefully to ensure that it handles cases near a “boundary”, such as an exact power of two or a number very near a power of two.

17.4.12. Because c(NUM) could be the maximum negative number, we should (in that case) consider the result in GR0 to be a number in the logical representation.

```
         L     0,NUM              Load number to be tested into GR0
         LPR   1,0                Move its magnitude to GR1, set CC
         L     0,=F'31'           Maybe exponent is 31?
         BO    Done               If the LPR overflowed, we're done
         SRL   0,5                Force c(GR0)=0 without CC setting
         BZ    Done               If CC=0, c(NUM) was zero
         L     0,=F'1'            Put starting power of 2 in GR0
ShiftR   SRA   1,1                Shift right once
         BZ    STOP               Stop if we lost most significant bit
         SLL   0,1                Double the power of 2
         B     ShiftR             And try again
Done     - - -                    C(GR0) = the positive power of 2
```

You may hve tried another approach: shift a "power-of-two" bit to the right, while shifting c(NUM) to the left and testing for overflow. (If you didn't, try it!)

17.4.13. The modifications are simple:

```
         L     1,NUM              Get number to be checked
         SR    0,0                Set exponent to 0
Test     SRA   1,1                Shift right once
         BZ    Done               Exit if now zero
         A     0,=F'1'            Add 1 to exponent
         B     Test               And try again
Done     - - -                    Exponent N now in GR0
```

17.4.14. It is always true for nonnegative register contents, and for negative values in which the rightmost N bits are zero. Other positive values will be forced to the next lower integer. For negative integers where nonzero bits are lost, we can see that the truncation is still "downwards" toward more negative values: consider a single right shift of X'FFFFFFFD', or −3. After the shift, the result is X'FFFFFFFE', which has value −2.

17.4.15. Testing for the possibility of overflow is more difficult than in Exercise 17.3.15, because we cannot simply examine the high-order bits remaining after shifting the desired part out of the register. Even if the bits to the left of the shifted part are all 0 or all 1, the most significant bit of the shifted part might be different, which would imply that the value was too large for the allotted field.

```
         LM    2,3,=F'0,1'        For setting data-fit flags
         L     0,Fourth           Get 4th integer
         LR    4,0                Save for overflow check
         STC   2,FLAG4            Set FLAG4 to 'fit' value
         SRDA  0,9                Position into packed word in GR1
         SLA   4,32-9             Test for fit
         BNO   OKFour             Skip if it fits correctly
         STC   3,FLAG4            Set FLAG4 to 'no-fit' value
OKFour   L     0,Third            Get third integer
         LR    4,0                Save for check
         STC   2,FLAG3            Set FLAG3 assuming a fit
         SRDA  0,13               Position in packed word
         SLA   4,32-13            Test for fit into 13 bits
         BNO   OKThree            Skip if it fits correctly
         STC   3,FLAG3            Set FLAG3 to 'no-fig' value
OKThree  - - -                    ...similarly for the other integers
```

We will see simpler ways to set indicator flags in Section 24.

17.4.16. Here is one way to do the bit count:

```
         SR    0,0                Clear GR0 for byte to be counted
         SR    2,2                Clear GR2 for bit count
         IC    0,XX               Pick up data byte
Loop     SR    1,1                Clear GR1 for shifting a bit
         SRDA  0,1                Shift a bit into GR1, set CC
         BZ    Store              Exit loop if no 1-bits left
         LTR   1,1                Check GR1 for a 1-bit
         BZ    Loop               Loop if it was a 0
         A     2,=F'1'            Increment bit count
         B     Loop               Continue testing
Store    STC   2,XX               Store bit-count back at XX
```

17.4.17. The values are

```
(1) X'FFFFFFEDCBA98765', CC=1
(2) X'8765432100000000', CC=3 (fixed-point overflow)
(3) X'FEDCBA9800001D95', CC=2 (the high-order half of GG0 is unchanged)
(4) X'00003FB72EA61D95', CC is unchanged.
```

17.4.18. The results are:

1. SRA 0,20 – c(GR0) = X'FFFFF876', CC=1
2. LPR 0,0 – c(GR0) = X'789ABCDF', CC=2
3. SLA 0,28 – c(GR0) = X'90000000', CC=3

17.4.19. This is one of many ways to do this:

```
         LA    0,8               Count bits in GR0
         LA    1,Char            Address of output characters
         ICM   3,B'1000',Byte    Insert the byte to be formatted
Loop     SR    2,2               Clear GR2
         SLDL  2,1               Shift a bit into GR2
         LA    2,X'F0'(0,2)      Make an EBCDIC character
         STC   2,0(,1)           Store the formatted digit
         LA    1,1(,1)           Step to next output character
         SH    0,=H'1'           Reduce bit count
         BP    Loop              Repeat for all 8 bits
```

17.4.20. We'll assume the shift count N in GR2 lies between 0 and 63; if not, we can first isolate the rightmost 6 bits of GR2.

```
         LTR   0,0               Check sign of c(GR0)
         SRDL  0,0(2)            Shift original argument by N bits
         BNM   NonNeg            Branch if original value nonnegative
         L     4,=F'-1'          Put 32 1-bits in GR4
         C     2,=F'32'          Check shift N count < 32?
         BL    ShortShf          Branch if shift count N < 32
         LR    0,4               Put 32 1-bits in GR0
         SRDL  4,32(2)           Shift N-32 bits into GR5
         ALR   1,5               Insert leading sign bits in GR1
         B     SetNegCC          Finish by setting Condition Code
ShortShf SRDL  4,0(2)            Shift N 1-bits into GR5
         ALR   0,5               Insert leading sign bits in GR0
SetNegCC LTR   0,0               Set CC=1
         B     Done
NonNeg   LTR   0,0               Check for nonzero bits in GR0
         BP    Done              CC=2 if c(GR0)>0
         LTR   1,1               Otherwise CC=0 or 2
Done     - - -                   Shifted result in c(GR0,GR1)
```

17.4.21. The solution is to check whether a one bit was shifted out; if so, add 1 to the shifted operand.

```
         L     0,=F'-5'          c(GR0) = X'FFFFFFFB' = -5
         SR    1,1               Clear GR1
         SRDA  0,1               C(GR0) = X'FFFFFFFD' = -3
         BNM   OK                OK if number was non-negative
         LTR   1,1               Test shifted bits
         BZ    OK                Some bit is not 1
         A     0,=F'1'           Add 1 to GR0, c(GR0) = -2
OK       - - -
```

Note that if the original value is even, the shifted bit is always zero. What happens if you start with −1? (See Exercise 17.4.14 also.)

17.4.22. Use an arithmetic shift (in either direction) with a zero shift amount.

17.4.23. 0 and −1.

Section 17.5

17.5.1. Here's a possible solution:

```
         LH    2,NN              Get shift count
         SRDL  2,5               Shift 5 bits right
         SR    2,2               Now set GR2 to zero
         SLDL  2,5               Rotate count (modulo 32) in GR2
         L     0,DataWord        Get data to be rotated (abc...xyz)
         LR    1,0               Copy it to GR1
         SLL   0,0(2)            Shift left N (cd...xyz00)
         LCR   2,2               -N in GR2
         SRL   2,32(2)           Shift right 32-N (00abc...x)
         ALR   0,1               Merge the parts (cd...xyzab)
         ST    0,RotateWd        Store the rotated result
```

17.5.2. Here, we can use SLDL to help:

```
         LH    2,NN              Get shift count
         SRDL  2,6               Shift 6 bits into GR3
         SR    2,2               Set GR2 to zero
         SLDL  2,6               Shift count (modulo 64) in GR2
         LM    0,1,DWData        Get operand to be rotated
         CH    2,=H'32'          Rotating more than 32 bits?
         BL    Under32           Branch if yes
*        Swap the halves, and then rotate N-32 bits
         LR    3,0               Copy high-order half to GR3
         LR    0,1               Move low-order half to GR0
         LR    1,3               And high-order half to GR1
         SH    2,=H'32'          Reduce count by 32
Under32  LR    3,0               Copy high-order half to GR3
         SLDL  0,0(2)            Shift both halves left N(mod 32) bits
         STM   0,1,RotatDWd      Store both parts of (temporary) answer
         LCR   2,2               Complement N (now within (-31,0))
         SRL   3,32(2)           Shift high-order part right 32-N bits
         AL    3,RotatDWd+4      Add low-order piece of result
         ST    3,RotatDWd+4      Store final low-order half of result
```

17.5.3. `RLLG  r,r,32`

Section 17.6

17.6.1. The statement will be assembled as though we had written

```
         SLL   9,0(2)
```

The contents of GR2 will depend on its position in the program, so the shift amount could vary with each assembly. (Not a good idea!) It's possible that some USING statements specifying the same relocatable value as **A** and a *higher* register number were available for resolving Addressing Halfwords, in which case that register would be used as the base register for the shift. (An even worse idea!)

17.6.2. Assuming normal base register usage conventions, the shift amount might be any even number from 0 to 62. In any given execution of the program, the number of shifts will be fixed, but it will depend on where the program was placed in memory. (Again, not a good idea!)

17.6.3. The number of shifts is *not* 12! As in Exercise 17.7.2, the number of shifts will be found when the address of **AAA** is evaluated; this could be any multiple of 4 between 0 and 60, because **AAA** is fullword aligned.

17.6.4. Be careful! If the original value in Rx is negative, SLDA and SRDA won't be able to set it to zero. Furthermore, if `x` is an odd-numbered register, none of the double-length shift instructions can be used. If `x` is even, SLDL and SLDA clear register x+1 instead, and SRDL and SRDA effectively move c(GRx) to c(GRx+1). Thus, (4), (6), (7), and (8) should be avoided.

17.6.5. Your table might look like this:

Shift	Direction	Length	Type	Mnemonic
S	L	S	L	SLL
S	L	S	A	SLA
S	L	D	L	SLDL
S	L	D	A	SLDA
S	R	S	L	SRL
S	R	S	A	SRA
S	R	D	L	SRDL
S	R	D	A	SRDA

17.6.6. There is no reason to shift more than 31 positions in a single register; the result can be obtained either with a shift of 31 or fewer positions, or with a different (and probably faster) instruction.

17.6.7. We will see later that the UNPK instruction provides a simpler solution to this problem.

```
         SR    2,2              Clear GR2 for data byte
         IC    2,DATA           Get 2 hex digits from data
         SRDL  2,4              Shift right digit into GR3
         IC    2,ChTbl(2)       Get EBCDIC character for left digit
         SRL   3,28             Position right digit for indexing
         IC    3,ChTbl(3)       Get EBCDIC character for right digit
         STC   2,CH             Store first character of result
         STC   3,CH+1           And second character also
         - - -
ChTbl    DC    C'0123456789ABCDEF'
```

17.6.8. The values by which the Effective Address should be divisible are:

1. L – 4
2. BC – 2 (it must be!)
3. LH – 2
4. ICM – 1
5. LR – Irrelevant: LR generates no Effective Address!
6. SRDA – 1
7. STM – 4
8. STC – 1

17.6.9. The shift amount depends on the Effective Address, so it could be any multiple of 4 between 0 and 60 (assuming the Assembler has placed the literal on a word boundary).

Section 17.7

17.7.1. The three statements will generate the same bit patterns, but only the constant named A will always be aligned on a word boundary.

17.7.2. The generated constant is X'D864E26F'. If the letter U is omitted, all four data items are too large for the allotted fields.

17.7.3. The generated constant is X'8C4417E1'.

17.7.4. You could write the constants as

```
         DS    0F
UnsdVals DC    FL.(LA)'U432',FL.(LB)'U12',FL.(LC)'U5001',FL.(LD)'U47'
SgndVals DC    FL.(LA)'-232',FL.(LB)'-8',FL.(LC)'-4001',FL.(LD)'-31'
```

17.7.5. Write the length modifier as L.(8*A+B)

Programming Problem 17.1.

This solution uses a logical left shift to move bits into the sign position, where they are tested with the LTR instruction. Can you think of a way to use an SLA instruction and a test for overflow instead? Modify this solution to test for several data values.

```
         Title 'Solution to Problem 17.1'
*        Count the number of 1-bits in a word
         Print NoGen
P17_1    START 0
         BASR  15,0                Set GR15 as base register
         USING *,15                Inform the assembler
         L     0,DATA              Get data from memory
         SR    1,1                 Set shift counter to zero
Loop     SLL   0,1                 Shift data left one bit
         LTR   0,0                 See if sign bit is a one
         BZ    Error               Error if zero, no one-bits there
         BM    Finish              Branch if yes, all done
         A     1,=F'1'             Add one to shift count
         B     Loop                And go shift the data again
*
Finish   STH   1,Count             Store the shift count
         SRL   0,1                 Move the data back once
         ST    0,Norm              Store shifted result
         PrintOut Data,Norm,Count,*,Header=NO  Print answers and stop
Data     DC    F'7294'             A number to be tested
Norm     DS    XL4                 Space for shifted result
Count    DS    H                   Space for final shift count
*
Error    PRINTOUT *                Error stop for bad data
         END   P17_1
```

The output for this sample Data item is:

```
 Data     =        7294
 Norm     = X'71F80000'
 Count    =          18
```

Programming Problem 17.3.

This solution uses the arithmetic shifts:

```
P17_3    Start 0
*        Unpack four signed integers packed in a word.
         Print NoGen
         Using *,15
LA       Equ   9                   Bit length of integer A
LB       Equ   4                   Length of B
LC       Equ   13                  Length of C
LD       Equ   6                   Length of D
         L     0,SgndVals          Get data word into GR0
         SRDA  0,LD                Shift 6 bits into GR1
         SRA   1,32-LD             Sign-extend to right
         ST    1,Fourth            Store fullword result D
         SRDA  0,LC                Shift off 13 more bits into GR1
         SRA   1,32-LC             Shift with sign extension
         ST    1,Third             Store signed result of C
         SRDA  0,LB                Shift off next 4 bits for B
         SRA   1,32-LB             Sign-extend second integer
         ST    1,Second            Store final result of B
         ST    0,First             Store correct first integer A
         PrintOut First,Second,Third,Fourth,*,Header=NO
First    DS    F
Second   DS    F
Third    DS    F
Fourth   DS    F
SgndVals DC    0F,FL.9'-232',FL.4'-8',FL.13'-4001',FL.6'-31'
         End
```

The printed output shows the desired values:

```
First    =        -232
Second   =          -8
Third    =       -4001
Fourth   =         -31
```

Section 18 Solutions

Section 18.2

18.2.1. For 32-bit operands using signed multiplication, the product of two maximum negative integers yields the 64-bit signed product $+2^{62}$ = X'40000000 00000000', or 4611686018427387904.

18.2.2. If the operands x and y are positive, the products XY and xy are identical. If x is negative and y is positive, the logical product XY is actually $(2^{32}+x)y$; thus we need to add Y to the high-order half of the product xy as computed by the M (or MR) instruction. Similarly, if both x and y are negative,

$$XY = (2^{32}+x)(2^{32}+y) = 2^{32}(x+y) + xy$$

The 2^{64} term is ignored in our 64-bit product, so we simply add x and y to the high-order half of the product xy.

```
         L     2,X               Get first logical operand
         L     1,Y               Get second logical operand
         LTR   3,1               Save for addition, and set CC
         MR    0,2               Form arithmetic product
         BNM   NonNeg1           Branch if 2nd operand not negative
         ALR   0,2               Add first operand to high-order half
NotNeg1  LTR   2,2               Check sign of first operand
         BNM   NotNeg2           Branch if arithmetic rep. not minus
         ALR   0,3               Add 2nd operand to high-order half
NotNeg2  STM   0,1,LogProd       Store 64-bit logical product
```

Thus:

- If $X < 0$, add $Y \times 2^{32}$
- If $Y < 0$, add $X \times 2^{32}$

18.2.3. The largest signed 32-bit magnitude is (-2^{31}), and the largest signed 16-bit magnitude is (-2^{15}), so the largest 48-bit product is 2^{46}=70368744177664. The largest signed 64-bit magnitude is (-2^{63}), so the largest 96-bit product is 2^{94}=19807040628566084398385987584.

18.2.4. Because the two operands are positive, the test is simple:

```
         L     1,A               Get multiplier
         M     0,B               Form product
         LTR   0,0               Test high-order half
         BNZ   Overflow          Error if not all zero
         LTR   1,1               Test leftmost bit of low-order half
         BM    Overflow          If a 1-bit, result is too large
```

18.2.5. In this case it is simplest to use SLDA for the test. Because the product must be left intact in (GR0,GR1), the shift must be done in a different register pair.

```
         L     1,A               Load multiplier
         M     0,B               Form product
         LR    2,0               Copy left and ...
         LR    3,1               Right halves to (GR2,GR3)
         SLDA  2,32              Now test for fit in one word
         BO    Overflow          If overflow, product won't fit
```

18.2.6.

```
         L     1,A               Load A into GR1
         M     0,B               Multiply by B in (GR0,GR1)
         LR    2,0               Copy high-order word to GR2
         LR    3,1               Copy low-orer word to GR3
         SLDA  2,32              Test for any overflow
         BNO   Okay              Branch if no overflow
         LTR   0,0               Test sign of original result
         BM    OverNeg           Branch if negative result is too big
         B     OverPos           Branch if positive result is too big
Okay     - - -                   Result is representable in a fullword
```

18.2.7. This solution references each pair of halfwords using the addresses in GR11 and GR12, incrementing them by 2 each time. The accumulating sum is in (GR0,GR1):

```
         L     2,=F'10'            Count items in GR2
         SR    0,0                 Accumulate sum in (GR0,GR1)
         LR    1,0                 Initialize sum to zero
Loop     LH    4,0(,11)            Get an operand
         MH    4,0(,12)            Multiply by the other
         SRDA  4,32                Extend word product to 64 bits
         ALR   1,5                 Add low-order words
         BC    12,NoCarry          Skip carry add if none to add
         AL    0,=F'1'             Add a carry bit
NoCarry  ALR   0,4                 Now add high-order words
         AL    11,=F'2'            Add 2 to address of first table
         AL    12,=F'2'            And to the address of second table
         S     2,=F'1'             Count down items by 1
         BP    Loop                Repeat if not done
         STM   0,1,DwSum           Store accumulated sum
```

18.2.8. Since x is always positive, we only need to test the sign of y:

```
         L     2,Y                 Get (possibly negative) 2nd operand
         LTR   1,2                 Move to GR1 and set CC
         M     0,X                 Multiply to form arithmetic product
         BNM   Y_NonNeg            Now skip if y is nonnegative
         AL    0,X                 Otherwise add correction term
Y_NonNeg STM   0,1,LProd           Store 64-bit logical product
```

18.2.9. Because the CPU must eventually access the even-numbered register, its "address" is needed anyway. Furthermore, it is consistent to require all instructions referring to a register pair to do so with the even-numbered register.

18.2.10. Only one instruction is needed:

```
         SLDA  0,32                Destroys original (GR0,GR1) contents
         BO    Overflow            Branch if overflow is indicated
```

18.2.11. The largest positive 32-bit magnitude is $(2^{31}-1)$, and the largest positive 16-bit magnitude is $(2^{15}-1)$, so the largest 48-bit product is 70366596661249. The largest positive 64-bit magnitude is $(2^{63}-1)$, so the largest 96-bit product is 19807040619342712359383728129.

18.2.12. This technique is used: form the 64-bit product of A and B; then, if an arithmetic test of B indicates that it's negative, add $B \times 2^{32}$ to the product.

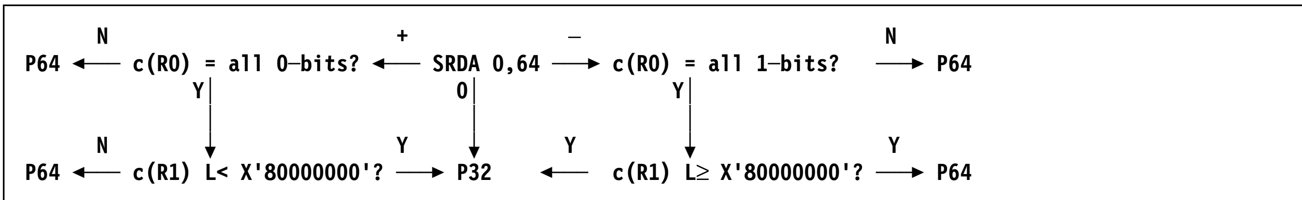
```
         L     1,A                 A in GR1
         L     0,B                 B in GR0
         LTR   0,0                 Test sign of B
         MR    0,0                 Form the product A*B
         BNM   Skip                Skip adding if B is not negative
         AL    0,B                 Complete the 64-bit product
Skip     DC    0H'0'
```

18.2.13. His instructions are incomplete. Consider multiplying 75141×56789: the product X'FE5808A9' is indeed 32 bits long but appears to be negative, −27785047. An additional test is needed:

```
         L     1,X                 Load first operand
         M     0,Y                 Multiply by second operand
         LTR   0,0                 Check high-order 32 bits
         BNZ   NotOK               If not zero, product is too big
         LTR   1,1                 Check high-order bit of GR1
         BZ    ProdOK              Branch if high-order 33 bits are 0s
         - - -                     Not OK
X        DC    F'75141'
Y        DC    F'56789'
```

x

18.2.14. In the following sketch, the characters "L<" and "L≥" mean "Logically Less Than" and "Logically Greater Than Or Equal", respectively



Some instructions to determine which of the two possible product fields should be used could be like these:

```
         SRDA  0,0                Check sign and possible zero
         BZ    P32                Product is zero, store as 32 bits
         BP    Plus               Go handle positive product
         C     0,=F'-1'           Is high half all 1-bits?
         BNE   P64                No, must store 64-bit product
         CL    1,=X'80000000'     Is low half logically < 2**31?
         BNL   P32                Yes, can store 32-bit product
         B     P64                Otherwise, store 64-bit product
Plus     LTR   1,1                Is low half all 0-bits?
         BNE   P64                If not, must store 64-bit product
         CL    1,=X'80000000'     Is low half logically >= 2**31?
         JL    P32                No, OK to store 32-bit product
P64      STM   0,1,Prod64         Store 64-bit product
         B     Done
P32      ST    1,Prod32           Store 32-bit product
Done     - - -
```

The solution above illustrates the general idea that testing only the high-order half of the product is not sufficient. This more elegant solution takes many fewer instructions:

```
         LR    2,1                Copy low-order half of product
         SRA   2,31               Copy sign bit through R2
         CR    2,0                Compare to high-order half of prod
         JNE   Prod64             If not equal, it's a 64-bit product
         ST    1,Prod32           Store the 32-bit product
         J     Done               Finish
Prod64   STM   0,1,Prod64         Store the full  64-bit product
Done     - - -
```

The key to this solution is that the SRA fills R2 with sign bits of the low-order half of the product, so R2 contains either 0 or −1. If that compares unequal to c(R0), we know there are significant bits in R0; and if it compares equal, we know that the sign bit of R1 is the same as the bits in R0 so the product has only 32 significant bits. Worth studying!

18.2.15. X'FFFFFFC0' = −64.

18.2.16. The values are (1) X'40000000' and (2) X'00000001' (the square of −1).

Section 18.3

18.3.1. For 32-bit operands using logical multiplication, the product of two 32-bit maximum unsigned integers ($2^{32}-1$, X'FFFFFFFF') yields the 64-bit unsigned product X'FFFFFFFE00000001', or 18446744065119617025.

18.3.2. Using the same analysis as in Exercise 18.2.2, we form an arithmetic product using logical multiply instructions:

```
         L     1,P                Load P into GR1
         L     2,Q                Load Q into GR2
         LTR   1,1                Set CC for P
         MLR   0,Y                Logical product of P and Q
         BNM   B1                 Branch if P >= 0
         SLR   0,2                Subtract Q, correct for negative P
B1       LTR   2,2                Check sign of Q
         BZ    B2                 Branch if Q >= 0
         SL    0,P                Subtract P, correct for negative Q
B2       STM   0,1,ArProd         Store signed product
```

Section 18.5

18.5.1. The range of QD values is $1 \leq QD \leq ND$, and the range of RD values is $1 \leq RD \leq DD$. (The lower bound of 1 is because zero is a one-digit number.)

Section 18.6

18.6.1. Because the quotient is represented as a signed two's complement number, its magnitude is less than or equal to 2^{31}. We might say that room must be left for the quotient's sign bit.

18.6.2. A specification error occurs if n is odd. If n is even, a divide-check interruption occurs *only* if the dividend is *not* a negative number between -1 and $-(2^{31}-1)$ and the divisor is -1. Thus it is highly probable that an interruption will occur. Here's a more detailed analysis:

- If $c(GR_n)$ is positive, the inequality in Figure 132 on page 277 is not satisfied.
- If $c(GR_n)$ is zero, an immediate interruption occurs.
- If $c(GR_n)$ is negative and not equal to -1, the inequality in Figure 132 on page 277 is not satisfied.
- If $c(GR_n)$ is negative and equal to -1, we must then consider the contents of $c(GR_n+1)$:
  - If $c(GR_n+1)$ is non-negative, the inequality in Figure 132 on page 277 is not satisfied.
  - If $c(GR_n+1)$ is negative, and equal to the maximum negative number, $(-2^{31}-1) \div (-1)$ fails because the quotient would be $+2^{31}$.
  - Any other negative value is satisfactory.

18.6.3. A fixed-point divide interruption can occur only if the divisor is zero. The quotient may be too large to fit in a halfword, but no error indication is given.

```
         L     0,WDividen        Fetch 32-bit dividend
         SRDA  0,32              Extend sign to 64 bits
         LH    2,HDivisor        Extend divisor to 32 bits
         DR    0,2               Divide
         STH   0,HRemaind        Store halfword remainder
         STH   1,HQuotent        And halfword quotient
```

18.6.4. A fixed-point divide exception is possible only if you divide the maximum negative number -2^{31} by -1, because the quotient $+2^{31}$ cannot be represented correctly.

18.6.5. A fixed-point divide interruption cannot occur: the dividend must be less than 2^{31} in magnitude, so division by 3 must yield a quotient whose magnitude is less than 2^{30}.

18.6.6. Because c(NN) is positive, there is no need to worry about a carry: at worst, a carry could propagate into the sign bit of the low-order half of the dividend. Since that bit is not a sign bit, no overflow need be signaled.

```
         L     7,NN              Get positive dividend
         SR    6,6               Set high-order half to zero
         AL    7,=F'5'           Add rounding factor logically
         D     6,=F'10'          Compute rounded quotient
         ST    7,QQ              And store the result
```

18.6.7. As in the solution to Exercise 18.6.6, it can be shown that subtracting 5 from the low-order half of a dividend cannot cause a borrow from the high-order register. Thus we could write the code sequence as follows:

```
         L     0,=F'5'           Set up rounding factor
         L     6,NN              Get signed dividend
         SRDA  6,32              Extend to 64 bits, and set CC
         BNM   Skip1             Branch if nonnegative
         LCR   0,0               Complement the rounding factor
Skip1    ALR   7,0               Add signed roundoff to right half
         D     6,=F'10'          Divide by 10
         ST    7,QQ              Store signed rounded quotient
```

18.6.8. Simulating logical division using arithmetic divide instructions is more difficult than it might first appear. It *can* be done, however. If you try, be sure to test your code sequence with many values representing “boundary” conditions.

18.6.9. This solution shows one way to do it:

```
         SR    0,0               Assume c(GR1) is nonnegative
         LTR   1,1               Check sign of c(GR1)
         BNM   Next              Branch if not negative
         L     0,=F'-1'          Fill GR0 with sign bits
Next     DR    0,2               Do the division
```

This exercise was provided purely for its educational (and entertainment) value; the coding technique is clumsy and unnecessary.

18.6.11. Because you can't represent 1/2 as an integer, you can revise the rounding formula like this:

```
(a)      quotient = ((2*dividend) + divisor) / (2*divisor)
    or
(b)      quotient = (dividend + (divisor/2)) / divisor
```

Your choice of formulas might depend on the magnitudes of dividend and divisor, and whether the divisor is even. Here's how you might code formula (a):

```
         L     0,Dividend         Assume a 32-bit dividend
         L     2,Divisor          Assume a 32-bit divisor
         AR    0,0                2*dividend
         AR    0,2                (2*dividend+divisor)
         SRDA  0,32               Extend to 64 bits for division
         AR    2,2                2*dividend
         DR    0,2                Rounded quotient now in GR1
```

We've assumed that magnitudes are small enough that no overflows will occur.

18.6.12. It's simpler to write

```
         SRA   5,2                Divide c(GR5) by 2
```

Because c(GR5) is nonnegative, SRL could also be used.

18.6.13. Let N, D, Q, and R represent the numerator, divisor, quotient, and remainder respectively. Then we want to calculate $Q = (N/D)+(1/2)$. We can rewrite this as $Q = (N+D/2)/D$. After dividing, if $2\times R \geq D$, we add 1 to Q. For positive N and D, we can write these instructions:

```
         L     2,N           Get numerator
         L     4,D           Get divisor
         LR    3,4           Copy to GR3
         SRA   3,1           Form D/2
         AR    2,3           Form modified numerator
         SRDA  2,32          Form 64-bit
         DR    2,4           Calculate tentative quotient in R3
         AR    2,2           Calculate 2*R
         CR    2,4           Compare to divisor
         BL    NoRound       If less, no rounding
         A     3,=F'1'       Round up the quotient
NoRound  - - -
```

Now, you should revise these instructions to handle N and D with arbitrary signs.

Section 18.7

18.7.1. Simulating arithmetic division using logical division instructions is more complex than a related simulation of arithmetic multiplication. This sequence of instructions handles most cases, but does not test for possible divide exceptions and does not preserve the Condition Code.

```
         LM    0,1,NumL           Get signed dividend in (GR0,GR1)
         LTR   3,0                Check dividend sign; save in GR3
         BNM   GetDivsr           Not negative, no need to complement
         LCR   0,0                Complement high-order dividend
         LCR   1,1                Complement low-order half
         BZ    GetDivsr           If low word zero, high word is OK
         S     0,=F'1'            Complemented dividend now +
GetDivsr L     2,DenL             Divisor in GR2
         LTR   4,2                Check divisor sign; save in GR4
         BNM   Divide             If not negative, skip complement
         LCR   2,2                Complement to make + divisor
Divide   DLR   0,2                Logical division, positive operands
         LTR   2,2                Check dividend sign
         BNM   PosDvdnd           Branch if nonnegative dividend
         LCR   0,0                Must have negative remainder
         LTR   4,4                - dividend; check sign of divisor
         BNM   CompQuot           Mixed signs, complement quotient
         B     Store              Both -, store result
PosDvdnd LTR   4,4                + dividend; check divisor sign
```

```
         BNM   Store              Both +, store result
CompQuot LCR   1,1                Complement quotient
Store    STM   0,1,RemQuot        Store arithmetic remainder/quotient
```

You may want to test this code sequence with some sample data, and compare the results of “normal” and simulated arithmetic division operations.

18.7.2. Both quotient and remainder are $2^{32}-2$, and the divisor is $2^{31}-1$. Thus, quotient×divisor = $2^{64}-2^{33}-2^{32}+2$; adding the remainder gives $2^{64}-2^{33}$, the dividend.

Section 18.8

18.8.1. Though ease of hardware implementation is important, the method you choose is up to you, and depends mainly on your habits and intuitive expectations. Consider the following examples:

dividend / divisor	System z		rem ≥ 0		rounded	
	quot.	rem.	quot.	rem.	quot.	rem.
+8 / +3	+2	+2	+2	+2	+3	−1
+8 / −3	−2	+2	−2	+2	−3	−1
−8 / +3	−2	−2	−3	+1	−3	+1
−8 / −3	+2	−2	+3	+1	+3	+1

Table 450. Examples of different types of integer division

1. With the System z rules, you expect the integer division of 8 by 3 to give quotient 2 and remainder 2, “except for signs”.
2. With a nonnegative remainder, the magnitudes of quotient and remainder behave less comfortably.
3. The quotient of ±8/±3 should be “near” 3, and the remainder will “fix it up”. The “rounded” rule might be chosen if the processor architecture represented all integer quantities in floating-point format.

18.8.2. Errors might occur in each case:

1. AR n,n will generate a fixed-point overflow if the result is not representable *and* an interruption if it is not masked off. This occurs if $c(GRn) \geq 2^{30}$ or $c(GRn) < -2^{30}$.
2. A specification error will occur if n is odd.
3. A specification error will occur if n is odd; if n is even, a fixed point divide exception always occurs either because $c(GRn,GRn+1) \geq 2^{32} \times c(GRn)$, or because c(GRn)=0.

Programming Problem 18.1.

We start with a value of x=F/2, and reduce x until a divisor is found.

```
        Title 'Solution to Problem 18.1'
*       Find prime divisors of N**3-1
P18_1   START 0
        BASR  15,0              Set base register
        USING *,15
        L     1,N               Get n
Outer   MR    0,1               n*n
        M     0,N               n*n*n
        S     1,=F'1'           F(n)
        LR    0,1               Initial x = F/2
        SRA   0,1               x in GR0
Inner   LR    3,1               Set up divide
        SR    2,2               Extend dividend
        DR    2,0               F(n)/x
        LTR   2,2               Check remainder
        BZ    Store             Branch if zero
        S     0,=F'1'           x=x-1
        B     Inner             Loop for next trial
Store   ST    0,X               Store x
        PrintOut N,X            Print values
        L     1,N               Pick up n again
        A     1,=F'1'           n=n+1
```

```
         ST    1,N               Restore n
         C     1,=F'8'           Test if done
         BNH   Outer             Loop if not for next n
         PrintOut *,Header=NO    Stop
N        DC    F'2'              Initial value of n
X        DS    F
         END   P18_1
```

Programming Problem 18.2.

The only real difficulty in this problem is computing the numerator function. In subtracting the final term 14, a logical subtract must be used to detect carry conditions correctly. The method used in this solution for computing the denominator avoids the possibility of the fixed-point overflow that will occur when computing 2*Xn-5 for the last time.

```
         Title 'Solution to Problem 18.2'
P18_2    START 0
         BASR  15,0              Set base register
         USING *,15
         LA    3,1               Xn in GR3
         LA    10,11             n in GR10
         SR    5,5               3*n in GR5
Begin    LR    4,3               Calculate denominator
         S     4,=F'5'           Xn-5
         AR    4,3               Xn + (Xn-5)
         LR    1,3               Now start on numerator
         A     1,=F'10727'       Xn + 10727
         SR    0,0               Clear GR0
         SLDA  0,0(5)            Xn*(Xn + 10727) in (GR0,GR1)
         SL    1,=F'14'          -14
         BC    3,Skip            Branch if a carry
         S     0,=F'1'           Else borrow one
Skip     DR    0,4               Quotient in GR1, remainder in GR0
         STM   0,1,REM           Store both for print
         ST    3,XN              Store Xn for print
         PrintOut XN,Quot,Rem    print results
         SLL   3,3               Form new Xn
         A     5,=F'3'           3*n
         S     10,=F'1'          Count down on number of tests
         BP    Begin             Loop for next n value
         PrintOut *,Header=NO    Terminate
Rem      DS    F
Quot     DS    F
XN       DS    F
         END   P18_2
```

Programming Problem 18.3.

By using an integer table containing values of length 1 byte, this solution can use the same numeric quantity to index through the list of primes and to count for termination.

```
         Title 'Solution to Problem 18.3'
P18_3    START 0
         BALR  12,0
         USING *,12
         LA    11,10             Number of primes p to test
         SR    10,10             Carry p in GR10
LoopA    IC    10,Table-1(11)    Get a prime from table
         ST    10,P              Store p for printing
         LA    9,1               Compute 2**p; set GR9 = 1
         SLL   9,0(10)           Shift left p places
         S     9,=F'1'           M(p) now in GR9
         ST    9,MP              Store M(p) for printing
         LA    7,4               Set up S1=4
         S     10,=F'2'          Set counter for p-2 calculations
LoopB    MR    6,7               Sn*Sn
```

```
         SL    7,=F'2'           Subtract two
         BC    3,Carry           Jump if a carry, no borrow
         S     6,=F'1'           Otherwise borrow from GR6
Carry    DR    6,9               Compute remainder modulo M(p)
         LR    7,6               Move back to GR7
         S     10,=F'1'          Count down number of multiplies
         BP    LoopB             Loop if still positive
         ST    7,Residue         Store final residue
         PrintOut P,MP,Residue   Print results
         S     11,=F'1'          Count down number of primes by 1
         BP    LoopA             Get next p
         PrintOut *,Header=NO    Terminate
F2       DC    F'2'              .*
Table    DC    FL1'31,29,23,19,18,13,11,7,5,3'  One-byte primes
P        DS    F
MP       DS    F
Residue  DS    F

         END   P18_3
```

Programming Problem 18.4.

```
         Title 'Solution to Problem 18.4'
P18_4    START 0
         BASR  15,0              Set base register
         USING *,15              Inform assembler
         SR    1,1               Initial value of X
*                                Assume worst case (denominator = 0)
Loop     L     2,Max             Set remainder
         LR    3,2               And quotient
*                                Calculate denominator
         LR    5,1               c(GR5) = X
         SH    5,=H'21'          Subtract 21
         MR    4,1               X*(X-21)
         AH    5,=H'131'         Add 131
         MR    4,1               X*(X*(X-21)+131)
         SH    5,=H'231'         Subtract 231
         BZ    Print             Go print if zero
*                                Calculate numerator
         LR    7,1               Move X to GR7
         MR    6,7               X squared in GR7
         LR    3,7               Move X**2 to GR3
         AH    3,=H'7'           Add 7
         MR    2,7               Multiply by X**2
         LR    2,3               Move to GR2 for shift
         SH    2,=H'11'          Subtract 11
         SRDA  2,32              Position for division
         DR    2,5               Divide by denominator
*                                Store for printing
Print    STM   1,3,X             Store X, Rem, Quot
         PrintOut X,Rem,Quot     Print results
         AH    1,=H'1'           Add 1 to X
         CH    1,=H'12'          Compare to 12
         BNH   LOOP              Branch if less or =
         PrintOut *,Header=NO    Terminate program
*                                Data areas
X        DS    F                 Value of argument X
Rem      DS    F                 Calculated remainder
Quot     DS    F                 Calculated quotient
Max      DC    F'-2147483648'
         END   P18_4
```

Programming Problem 18.5.

The largest value printed is 12 factorial = 479001600

```
         Print NoGen
P18_5    Start 0
         Using *,15
         LA    1,1                  Start with 1!
Loop     L     3,NFact              Get current value
         ST    1,N                  Store value of N
         MR    2,1                  Form N!
         ST    3,NFact              Store new value
         SLDA  2,32                 Check for overflow
         JO    Done                 If overflow, all done
         PrintOut N,NFact           Print values
         LA    1,1(,1)              Increment N
         J     Loop                 Repeat
Done     PrintOut *,Header=No       Stop
N        DS    F                    Current value of N
NFact    DC    F'1'                 0! = 1
         End   P18_5
```

Programming Problem 18.6.

Note that all arithmetic is done in the general registers!

```
        Title 'Solution to Problem 18.6'
P18_6   START 0
*       Calculate date of Easter for a specified year.
        Using *,15
* Step (1)
        LHI   0,19              Step (1) divisor
        L     2,Year            Get year
        SRDA  2,32              Extend to 64 bits
        LR    5,3               Copy to GR5
        LR    4,2               Copy high-order zeros to GR4
        DR    2,0               Calculate Y/19, A in GR2
* Step (2)
        LHI   0,100             Step (2) divisor
        DR    4,0               B in GR5, C in GR4
* Step (3)
        LR    6,5               Copy B to GR6
        SRDL  6,2               D in GR6
        SRL   7,30              E in GR7
* Step (4)
        LHI   0,25              Step (4) divisor
        LR    8,5               Copy B to GR8
        SLA   8,3               8B in GR8
        AHI   8,13              8B+13 in GR8
        SRDL  8,32              Extend to 64 bits
        DR    8,0               G in GR9
* Step (5)
        LHI   0,30              Step (5) divisor
        LR    10,2              Copy A to GR10
        MHI   10,19             19A
        AR    10,5              19A+B
        SR    10,6              19A+B-D
        SR    10,9              19A+B-D-G
        AHI   10,15             19A+B-D-G+15
        SRDL  10,32             Extend to 64 bits
        DR    10,0              H in GR10
* Step (6)
        LHI   0,319             Step (6) divisor
        LR    3,10              Copy H to GR3
        MHI   3,11              11H
        AR    3,2               A+11H
        SR    2,2               Clear GR2
        DR    2,0               M in GR3
* Step (7)
        SRDL  4,2               Divide C by 4
        SRL   5,30              J in GR4, K in GR5
* Step (8)
        LHI   0,7               Step (8) divisor
        AR    7,7               2E in GR7
        AR    7,4               2E+J
        AR    7,4               2E+2J
        SR    7,5               2E+2J-K
        SR    7,10              2E+2J-K-H
        AR    7,3               2E+2J-K-H+M
        AHI   7,32              2E+2J-K-H+M+32
        SR    6,6               Clear high-order 32 bits
        DR    6,0               L in GR6
* Step (9)
        LHI   1,25              Step (9) divisor
        LR    0,10              H in GR0
        SR    0,3               H-M
```

```
         AR    0,6                 H-M+L
         LR    5,0                 H-M+L in GR5
         AHI   5,90                H-M+L+90
         SR    4,4                 Clear high-order 32 bits
         DR    4,1                 N in GR5
         ST    5,Month
* Step (10)
         AR    0,5                 H-M+L+N in GR0
         AHI   0,19                H-M+L+N+19
         SRDL  0,5                 Divide by 32
         SRL   1,27                P in GR1
         ST    1,Day
         Printout Year,Month,Day,*,Header=NO
Year     DC    F'2010'
Month    DS    F
Day      DS    F
         END   P18_6
```

Programming Problem 18.7.

```
         Title 'Problem 18.7: Create a hexadecimal addition table'
         Print Nogen
P18_7    Start 0
         Using *,15                Establish addressability
         PrintLin HeadLine,L'HeadLine  Print heading line
         SR    0,0                 Initialize row value in GR0
         L     6,=A(X'F')          Initialize digit mask
Col_Loop SR    1,1                 Initialize column value in GR1
         LR    2,1                 Copy column value to GR2
         IC    2,Chars(2)          Get its EBCDIC representation
         STC   2,Line+1            Store in the print line
         L     3,=A(Line+3)        Initialize product-value position
Row_Loop LR    5,1                 Copy column value
         AR    5,0                 Multiply by row value
         LR    4,5                 Copy product to GR4
         SRL   4,4                 Move to rightmost 4 bits
         NR    4,6                 Leave high-order digit in GR4
         NR    5,6                 Leave low-order digit in GR5
         IC    4,Chars(4)          Get EBCDIC char for high digit
         IC    5,Chars(5)          And for low digit
         STC   4,0(,3)             Store high digit in print line
         STC   5,1(,3)             Store low digit in print line
         A     3,=F'3'             Step to next print-line position
         A     1,=F'1'             Increment column value
         C     1,=F'16'            Check for finished with this line
         BL    Row_Loop            If not done, repeat for next column
         PrintLin Line,L'Line      Print a line of the table
         A     0,=F'1'             Step to next row
         C     0,=F'16'            Check if all rows done
         BL    Col_Loop            If not done, repeat for next row
         Printout *,Header=NO      Terminate the program
Chars    DC    C'0123456789ABCDEF'     EBCDIC characters
HeadLine DC    C'1   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F'
Line     DC    CL(L'Headline)' '
         End   P18_7
```

Programming Problem 18.8.

```
         Title 'Problem 18.8: Create a hex multiplication table'
         Print Nogen
P18_8    Start 0
         Using *,15                Establish addressability
         PrintLin HeadLine,L'HeadLine  Print heading line
         SR    0,0                 Initialize row value in GR0
         L     6,=A(X'F')          Initialize digit mask
Col_Loop SR    1,1                 Initialize column value in GR1
         LR    2,1                 Copy column value to GR2
         IC    2,Chars(2)          Get its EBCDIC representation
         STC   2,Line+1            Store in the print line
         L     3,=A(Line+3)        Initialize product-value position
Row_Loop LR    5,1                 Copy column value
         MR    4,0                 Multiply by row value
         LR    4,5                 Copy product to GR4
         SRL   4,4                 Move to rightmost 4 bits
         NR    4,6                 Leave high-order digit in GR4
         NR    5,6                 Leave low-order digit in GR5
         IC    4,Chars(4)          Get EBCDIC char for high digit
         IC    5,Chars(5)          And for low digit
         STC   4,0(,3)             Store high digit in print line
         STC   5,1(,3)             Store low digit in print line
         A     3,=F'3'             Step to next print-line position
         A     1,=F'1'             Increment column value
         C     1,=F'16'            Check for finished with this line
         BL    Row_Loop            If not done, repeat for next column
         PrintLin Line,L'Line      Print a line of the table
         A     0,=F'1'             Step to next row
         C     0,=F'16'            Check if all rows done
         BL    Col_Loop            If not done, repeat for next row
         Printout *,Header=NO      Terminate the program
Chars    DC    C'0123456789ABCDEF'     EBCDIC characters
HeadLine DC    C'1   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F'
Line     DC    CL(L'Headline)' '
         End   P18_8
```

Programming Problem 18.9. This solution uses a symbolic constant **ND** to define the number of terms in the sequence, and the number of fraction digits to be printed. The suggested limit of 115 digits is simply because the result displayed by the PrintLin macro is limited by the length of a print line to 121 bytes.

```
P18_9    CSect ,                   Calculate E to ND digits (ND < 115)
         Print NoGen
         Using *,15
ND       Equ   60
         XR    1,1                 Output digit index
         LA    2,ND                Counter for digit index
A        XR    6,6                 Q = 0
           LA    3,ND-1            Count K
           LA    4,1(,3)           K
B            XR    7,7             Inner loop to generate a digit
             IC    7,N(3)          Get working fraction R(K)
             MHI   7,10            10*R(K)
             AR    7,6             10*R(K)+Q(K+1)
             XR    6,6             Clear high-order dividend
             DR    6,4             Divide by K
             STC   6,N(3)          Save new R(K)
             LR    6,7             Copy Q(K)
             BCTR  4,0             K=K-1
             JCT   3,B             Repeat inner loop
           LA    6,X'F0'(,6)       Make a zoned digit
           STC   6,D(1)            Store in output string
           LA    1,1(,1)           Step output-digit index
           JCT   2,A               Repeat for next digits
         PrintLin V,L'V+ND         Print the result
```

```
         PrintOut *,Header=NO
N        DC    (ND)X'1'          Initial fraction numerators
V        DC    C'-E=2.'
D        DC    (ND)C' '          Digit string
         End   P18_9
```

The printed output is:

```
-E=2.71828182845904523536028747135266249775724709369995957496696 7
```

By choosing a larger size for each array element, you can make the number of terms greater; and by multiplying by a larger power of 10 you can generate multiple fraction digits on each iteration. You might enjoy using (for example) halfword terms and generating four fraction digits on each iteration. You'll have to work out how best to display the result on multiple print lines.

Programming Problem 18.10. This solution is not a very efficient way to search for prime numbers: test divisors need not be larger than the largest number less than the square root of the tested value.

```
         Print NoGen
P18_10   CSect ,
MaxTest  Equ   99                Largest Odd value to test
         Using *,15
         LA    2,2
         PrintOut Prime,Header=No Print the only even prime
         LA    2,3
         STH   2,Prime
         PrintOut Prime,Header=No Print the smallest odd prime
LoopA    LA    2,2(,2)           Generate next test value
         LA    3,3               Initial test divisor
LoopB    LR    4,2               Copy test number to GR4
         SRDA  4,32              Position test number for division
         DR    4,3               Divide by test divisor
         LTR   4,4               Test remainder
         BZ    NotPrime          Zero remainder, not prime
         LA    3,2(,3)           Next test divisor
         CR    3,2               Test for enough tests done
         BL    LoopB             Try again if test divisor not too big
         STH   2,Prime
         PrintOut Prime,Header=No Print the next odd prime
NotPrime C     2,=A(MaxTest)     See if test value is too big
         BL    LoopA             Repeat if not
         PrintOut *,Header=No    Stop
Prime    DC    H'2'              Initial value
         End   P18_10
```

Section 19 Solutions

Section 19.2.

19.2.1. I can't think of a good reason. Maybe they thought it wouldn't happen often enough?

Section 19.3.

19.3.1. This can be done using the instruction

```
         N     8,=A(X'01FFFFFF')  First seven mask bits are zero
```

We will see in Section 21 that newer instructions can eliminate the need for a mask operand in memory!

Section 19.4.

19.4.1. The only important feature of the solution is the logical AND instruction. Since we are shifting in a 64-bit register pair, we must guard against the possibility that a shift amount greater than 31 is specified, since this would lead to loss of bits when the SLDL is executed. Thus, we mask off all but the rightmost 5 bits of the shift amount. (A circular shift of 32 bits produces the original argument.)

```
         SR    0,0               Clear high-order register, GR0
         L     1,Data            Get word to be shifted into GR1
         L     2,NShifts         Load shift count into GR2
         N     2,=A(X'1F')       Mask shift count to 5-bit value
         SLDL  0,0(2)            Shift by required amount
         OR    1,0               Rotate high bits to bottom end
         ST    1,Data            Store rotated data
```

19.4.2. A simple solution follows from the observation that a 32-bit circular right shift of N places is the same as a circular left shift of 32−N places.

```
         SR    0,0               Clear GR0
         L     1,Data            Get data to be shifted into GR1
         L     2,NShifts         Get the shift count in Gr2
         LTR   2,2               Check sign of shift
         BNM   GoAhead           Proceed if nonnegative
         LCR   2,2               Form -N
         A     2,=F'32'          And now 32-N
GoAhead  N     2,=A(X'1F')       Mask shift amount to 5 bits
         SLDL  0,0(2)            Shift by required amount
         OR    1,0               Rotate high bits to bottom end
         ST    1,Data            Store rotated data
```

Alternatively, and maybe more obviously:

```
         - - -                   Initialize GR0, GR1, and GR2
         LTR   2,2               Check sign of N
         BNM   Go                Proceed if not negative
         SLDL  0,32              Exchange GR0 and GR1
         N     2,=A(X'1F')       Mask shift amount to 5 bits
         SRDL  0,0(2)            Shift right the specified amount
         B     Finish            And go complete the result
Go       N     2,=A(X'1F')       Mask for left shift
         SLDL  0,0(2)            Shift left the specified amount
Finish   OR    1,0               Complete the circular shift
         ST    1,Data            Store the rotated answer
```

19.4.3. The register contents are unchanged, and the CC setting indicates a zero or a nonzero result. The LTR instruction sets the CC for nonzero results depending on the sign, so the methods are not strictly equivalent, because NR and OR will not set the CC to 2.

Section 19.5

19.5.1. Consider all four possible bit combinations:

```
  Initially     After XR  1,2      After XR  2,1      After XR  1,2
c(GR1) = 1100       0110               0110               1010
c(GR2) = 1010       1010               1100               1100
```

and the two bit patterns have been interchanged.

This can't be done in the same way between registers and memory, because there is no Exclusive OR instruction that operates between operands in memory and in a register, *and* leaves the result in memory.

19.5.2. Useless and unintelligible junk.

19.5.3. Consider a 2-bit register (no pun intended), and the four possible values 00, 01, 10, and 11 for A and B. Make a table with 16 rows corresponding to the combinations of values of A and B, and columns for A, B, A+B, A AND B, (A+B) − (A AND B), A OR B, (A OR B) − (A AND B), and A XOR B. It works!

19.5.4. XOR is commutative — that is, (A XOR B) is always the same as (B XOR A). The order of the operands doesn't affect the final result, which is B in each case.

19.5.5. It works! The reasoning is the same as in Exercise 19.5.4:

```
         XR    1,0               means Old XOR New
         XR    0,1               means Old XOR (Old XOR New)
```

so the final result is the same as “New” because (Old XOR Old) is zero!

19.5.6. The second use of `=F'1'` can be replaced by `SL  8,=F'-1'`.

Now: if we replace the preceding AL by `SL  9,=F'-1'`, do we need to worry about the following BC? If you recall that the subtraction is performed by adding the ones' complement of the second operand (that will be zero) and a low-order 1-bit to the first operand, we see that nothing need be changed in the other instructions.

19.5.7. This statement also eliminates the need for the DS 0F:

```
         DC    A(X'7FFC0')
```

Section 19.6

19.6.10. These instructions need only as many iterations as there are numbers of 1-bits in c(GR0).

```
         SR    2,2               Set count of 1-bits to zero
A        LTR   1,0               Test if any 1-bits of X are left
         JZ    Done              Finished if none are left
         BCTR  1,0               Form X-1
         NR    0,1               Eliminate a 1-bit
         JCT   2,A               Count down and test for completion
Done     LCR   2,2               Count of 1-bits now in GR2
```

19.6.11. Yes. Consider $X=2^{31}-1$`X'7FFFFFFF'`; then X+1 is `X'80000000'`. Try the same for $X=2^{32}-1$.

19.6.12. If X=-1, the mask is zero: (NOT X) is zero, and (X+1) is also zero, so their AND is necessarily zero.

19.6.13. If X=0, the mask is all 1-bits.

19.6.14. If X=0, the mask is all 1-bits: (X-1) is all 1-bits, and XORing that with zero produces all 1-bits.

Section 19.7

19.7.1. Start with A=B'1010' and B=B'1100', and see what results from NOT(A), NOT(B), AND(A,B), OR(A,B), and XOR(A,B). The tables below show some combinations; note that not all combinations appear to be possible.

- Simulating AND(A,B):

OR and NOT	OR and XOR	XOR and NOT
NOT((NOT A) OR (NOT B))	(A OR B) XOR (A XOR B)	

- Simulating OR(A,B):

AND and NOT	XOR and NOT	XOR and AND
NOT((NOT A) AND (NOT B))	(A XOR B) XOR NOT(A XOR B)	(A AND B) XOR (A XOR B)

- Simulating XOR(A,B):

AND and OR	AND and NOT	OR and NOT
Not possible: (1 AND 1) and (1 OR 1) can never be 0.	NOT(NOT((NOT A) AND B) AND (NOT(A AND (NOT B))))	(NOT(A OR (NOT B))) OR (NOT((NOT A) OR B))

- Simulating NOT(A): remember that (NOT 0) is always 1, but none of (0 AND 0), (0 OR 0), and (0 XOR 0) can ever return 1. Thus there's no way to simulate NOT using any two of the other logical operators.

Programming Problem 19.1. This solution does not calculate the CC value for an arithmetic sum.

```
         Print NoGen
P19_1    Start 0
RCar     Equ   10                Carries kept in GR10
RAop     Equ   4                 A operand in GR4 (even)
RA       Equ   5                 'A' bit kept in GR5 (next odd)
RBop     Equ   6                 B operand kept in GR6 (even)
RB       Equ   7                 'B' bit kept in GR7 (next odd)
RCC      Equ   12                Condition code formed in GR12
RSum     Equ   8                 Sum developed in GR8
Rct      Equ   11                Count reg for number of bits
RT       Equ   0                 Temporary register
*
         Using *,15
         LM    1,3,DataPtr       Set loop registers for cases
Start    SR    RCC,RCC           Initial CC set to 0
         LR    RCar,RCC          Initial carry = 0
         LR    RSum,RCC          Initial sum = 0
         L     RAop,Data(1)      Initial 'A' operand
         L     RBop,Data+4(1)    Initial 'B' operand
         LA    Rct,32            Initialize bit counter
         ST    RAop,A            Save A for printing
         ST    RBop,B            Same with B
*
Inner    SRDL  RAop,1            Shift out the 'A' bit
         SRL   RA,31             Position at right end
         SRDL  RBop,1            Shift out 'B' bit
         SRL   RB,31             Position at right end
         OR    RSum,RA           Insert the new 'a' bit
         XR    RSum,RB           Exclusive or with 'B' bit
         XR    RSum,RCar         And with old carry bit.
         OR    RCC,RSum          Use sum for zero/nonzero CC
         SRDL  RSum,1            Move sum bit 1 place to right
         LR    RT,RCar           Compute new carry bit
         NR    RT,RA             (Old carry) and ('A' bit)
         NR    RA,RB             ('A' bit) and ('B' bit)
         NR    RCar,RB           ('B' bit) and (old carry)
         OR    RCar,RA
         OR    RCar,RT           New carry bit now complete
         S     Rct,=F'1'         Reduce count
         BP    Inner             Process next bit
         ALR   RCar,RCar         Double final carry bit
         OR    RCC,RCar          Pack into condition code
         STC   RCC,CCL           Store for printing
         ST    RSum+1,Sum        Store sum for printing
         PrintOut A,B,Sum,CCL,Header=No   Print results
         AR    1,2               Increment data index
         CR    1,3               Compare to last address
         BNH   Start             Continue with next data pair
         PrintOut *,Header=No    Terminat program
*
Data     DS    0D                Start of test cases
         DC    X'FFFFFFFF,00000001' Sum    zero,   carry: LCC=2, ACC=3
         DC    X'11111111,12345678' Sum nonzero, nocarry: LCC=1, ACC=2
         DC    X'00000000,00000000' Sum    zero, nocarry: LCC=0, ACC=0
```

```
         DC    X'FEDCBA98,12345678' Sum nonzero,   carry: LCC=3, ACC=1
EnData   Equ   *
DataPtr  DC    A(0,8,EnData-Data-8)
Sum      DS    XL4                 Final sum
A        DS    XL4                 Initial A operand
B        DS    XL4                 Initial B operand
CCL      DC    X'FF'               Logical-Add Condition Code
         END   P19_1
```

Programming Problem 19.2.

In we first initialize S(0) to A and 2*C(0) to B, we are ready to enter the loop. The sum is accumulated in GR1, carries are “carried” in GR3, and GR4 holds the working CC setting.

```
         SR    4,4                 Cet CC to 'zero, no carry' condition
         L     1,A                 Initialize GR1 to S(0)
         L     3,B                 initialize GR3 to 2*C(0)
Loop     LR    2,1                 Move S(n) to temporary in GR2
         XR    1,3                 Form S(n+1) in GR1
         NR    3,2                 Form C(n+1)
         ALR   3,3                 'Shift' carries left by 1 position
         BC    8,Done              If zero and no lost bit, we're done
         BC    4,Loop              If nonzero and no lost bit, repeat
         L     4,=F'2'             A bit carried out: set partial CC
         BC    1,Loop              If more carries to do, repeat
Done     ST    1,Sum               Store final sum
         LTR   1,1                 Check for nonzero result
         BZ    StoreCC             Jump if sum is zero, CC is okay
         A     4,=F'1'             Set CC to indicate nonzero sum
StoreCC  ST    4,CCodeL            Store final Condition Code
```

Determining the CC setting for arithmetic addition is left as an additional exercise for you.

Programming Problem 19.4.

The value NMax is defined so that the number of values tested is a multiple of 8; this allows proper packing of the results in the bit string at PrimeBts.

```
         Print NoGen
P19_4    CSect ,
N        Equ   400                 Maximum number to test
NMax     Equ   ((N/8)*8)           Round to a multiple of 8
Mark     Equ   0                   Mark composites with zero
         Using *,15
         LA    0,NMax              Maximum value to test
         LA    1,2                 Starting value
MarkLoop LR    2,1                 Stepping increment
         LA    3,0(2,2)            Starting index
         CR    3,0                 Check for end of table
         JNL   MarkDone            Done with marking if so
Marking  LA    4,Table-1(3)        Address of a Table entry
         MVI   0(4),Mark           Mark the entry as composite
         AR    3,2                 Step to next
         CR    3,0                 Test for past end of table
         JL    Marking             Repeat if not done
FindNext LA    1,1(,1)             Increment starting value
         CR    1,0                 Done all values?
         JNL   MarkDone            Branch if yes
         LA    4,Table-1(1)        Point to next position in Table
         CLI   0(4),Mark           Check for marked position
         JE    FindNext            This one marked, look again
         J     MarkLoop            Try next values
MarkDone LA    4,PrimeBts          Address of bit string
         LA    1,Table             Address of marked table entries
ByteLoop LA    2,8                 Count bits packed in a byte
```

```
         SR    3,3                Bits packed in GR3
PackLoop SLL   3,1                Make room for next bit
         CLI   0(1),Mark          Was this byte marked?
         JE    NotPrime           Skip bit insertion if yes
         LA    3,1(,3)            Insert a 1-bit for a prime
NotPrime LA    1,1(,1)            Point to next byte for mark test
         S     2,=F'1'            Count down bits in the byte
         JP    PackLoop           Repeat for all 8 bits
         STC   3,0(,4)            Store the packed byte in the string
         LA    4,1(,4)            Step to next packed byte position
         S     0,=F'8'            Count down number of bytes tested
         JP    ByteLoop           Repeat if more bytes to be packed
         PrintOut PrimeBts,*,Header=No Show the results
Table    DC    (NMax)X'1'         Initialize all to 'prime'
PrimeBts DC    XL(NMax/8)'0'      Prime bits
         End   P19_4
```

The output is:

```
PrimeBts = X'EA28A20A08A20828228220808A28800220A00A08220828028A00200228828020820A08A00800228800208028820208220809'
```

Programming Problem 19.5.

The modifications to the solution to Problem 19.4 are simple; the output is:

```
PrimeBts = X'F6D32D265948B6814C325261B0416984932C205A0486912522'
```

The largest prime representable in a string of 2^{30} bits would be less than 2^{34}..

Section 20 Solutions

Section 20.1

20.1.1. The generated Effective Addresses are:

1. X'0174629A'
2. X'01749816'
3. X'0172629C'
4. X'0174629E'

20.1.2. The generated Effective Addresses are:

1. X'007B1EEE'
2. X'007D68D2'
3. X'00734EC4'
4. X'007B1EF6'

20.1.3. The generated Effective Addresses are:

1. X'9610C63C'
2. X'9610D2FC'
3. X'9610C640'
4. X'9610A63A'

20.1.4. The IA has already been incremented by the length of the current instruction, which could be either 4 or 6 bytes long.

20.1.6. The respective Effective Addresss are:

(1) X'27B9B4'
(2) X'27B976'
(3) X'2839CC'
(4) X'26B9AE'

20.1.7. You can't. The address of an instruction is always even, and $2 \times I_2$ is also even.

20.1.8. This might better be called “offset” addressing. It's a risky technique, because something might be inserted between the symbolic target A or * and whatever is at the offset from that location; finding and correcting such usage is tedious and error-prone.

Section 20.2

20.2.1. The generated Effective Address in 24-bit mode is X'00360A', in 31-bit mode is X'0200360A', and in 64-bit mode is X'000000008200360A'.

20.2.2. The generated Effective Addresses are shown in the following table:

No.	24-bit mode	31-bit mode	64-bit mode
1	X'10C63C'	X'1610C63C'	X'000000009610C63C'
2	X'10D2FC'	X'1610D2FC'	X'000000009610D2FC'
3	X'10C640'	X'1610C640'	X'000000009610C640'
4	X'10A63A'	X'1610A63A'	X'000000009610A63A'

Section 20.3

20.3.1. LARL always adds its address to $2 \times I_2$, so the value in R_1 depends on the address of the LARL itself!

20.3.2. If we write `LA  reg,0(0,reg)` we get the same effect: in 24-bit addressing mode, the high-order byte is set to zero. The value of “reg” cannot be zero for the LA (why?). It may make a difference in some cases that the N instruction sets the condition code, but LA leaves it unchanged.

20.3.3. No, because the second operand appears to be an implied indexed address: **A** may not be a defined symbol, and **NoErr** is relocatable and therefore invalid as an index register specification.

20.3.4. No, because GR9 will contain the address of the literal, not the address of **NoErr**. Yes, if the LA is replaced by L *and* the literal is addressable.

20.3.5. The differences depend on three factors: whether **NoErr** is addressable, whether the literal is addressable, and the current addressing mode.

1. If *both* are indeed addressable, the results will be equivalent. This is true whether **NoErr** is absolute or relocatable. LA is preferable if **NoErr** is addressable, since it saves the four bytes required for the literal and the extra execution-time memory reference.
2. The second form is necessary if **NoErr** is not addressable. They are not equivalent if the literal is not addressable.
3. A-type constants don't depend on the addressing mode, while the LA and LAY instructions do. Normally this won't be a concern.

20.3.6. The claim is true. You could also write this code sequence:

```
         LA    8,X'101'(,5)
         SRL   8,0(8)
```

It's worth understanding how this works!

20.3.7. Using LA to increment general register contents is limited to

(1) values of "number" between 0 and 4095, *and* to
(2) nonzero values of GRx, *and*
(3) depends on the current addressing mode. For AMode 24, the sum cannot exceed $2^{24}-1$, and for AMode 31, the sum cannot exceed $2^{31}-1$. There is no limit on the sum for AMode 64.

If "number" is defined in an EQU statement, the halfword literal can't be used. (But, you could then use =Y(number) instead!)

In each case, it's possible for the sum to exceed the given limit and "wrap around" to zero. Consider incrementing c(GR1)=X'00FFFFFF' by 2 in 24-bit addressing mode: the Effective Address is 1!

In most cases, these three restrictions are satisfied. The CC setting is rarely interesting in such situations, so there is little reason to use AH. Also, the two bytes for the literal and an execution-time memory reference are saved.

20.3.8. The object code from the assembled instructions looks like this:

```
E300 0120 7A71       LAY   0,500000
E310 0EE0 8571       LAY   1,-500000
```

In the second instruction, you will see that the high-order bit of DH is X'80', indicating a negative long displacement.

If the addressing mode is 24- or 31-bit mode, the result of the second LAY instruction will be X'00F85EE0' or X'7FF85EE0', respectively: neither is −500000! You will get the intended negative value *only* if the addressing mode is 64-bit mode.

20.3.9. In the first case, the initial c(GR6) is X'00FFFFFF' and the result will be +1, because the generated Effective Address X'01000001' retains only the low-order 24 bits. In the second case, the generated Effective Address will be X'00FFFF02'.

20.3.10. Unless the LAY instruction is executed in 64-bit addressing mode, the Effective Address in GR3 will be X'00FFFFFF' in 24-bit mode, and X'7FFFFFFF' in 31-bit mode.

20.3.11. The results depend on the contents of the two registers. If c(GRy) is a non-negative integer with value less than 2^{24}, they are equivalent. Otherwise, the LA instruction will truncate significant high-order bits if the value is either negative or greater than or equal to that value, for addressing modes 24 and 31. For addressing mode 64, they are (almost) equivalent except that LA will change the high-order 32 bits of GGx.

20.3.12. You could use something like the following:

```
         LARL  0,BList            Address of branch list
         AR    15,0               Form address of selected branch
         BR    15                 Branch to the correct one
         - - -
BList    J     A
         J     B
         J     C
         J     D
```

None of the instructions requires a base register.

20.3.13. Remember: specifying register zero as a base or index register means "no register"!

20.3.14. In 24-bit addressing mode, the results in GR10 are:

1. X'00001234'
2. X'0000ABCD'

and in 31-bit addressing mode, the results in GR10 are

1. X'00001234' (the same)
2. X'007FABCD'

20.3.15. The Effective Addresses in 24-bit and 31-bit addressing modes are:

1. X'0000FE'(24), X'000000FE'(31)
2. X'01831B'(24), X'1B01831B'(31)
3. X'000009'(24), X'01000009'(31)

20.3.16. The resulting Effective Addresss are:

(1) No Effective Address can be generated, because the instruction cannot be addressed by the IA in the PSW! The CPU will attempt to fetch an instruction at address X'003B6D0E'.
(2) The Effective Address is X'543B6D0E'.
(3) The Effective Address is X'00000000543B6D0E'.

Section 21 Solutions

Section 21.1

21.1.1. The operand 76543 is too large for a halfword operand, so the instruction can't be assembled without error. You can use either of these instructions instead:

```
         LGFI  0,76543            ...clears high-order half of GG0
         IILF  0,76543            Insert operand into GR0
```

In each case, the assembled I_2 operand is X'00012AFF'.

21.1.2. The difference is in notation: an I_2 immediate operand is used in an arithmetic or logical operation, while an RI_2 immediate operand is used in a relative-offset instruction such as a branch.

Section 21.2

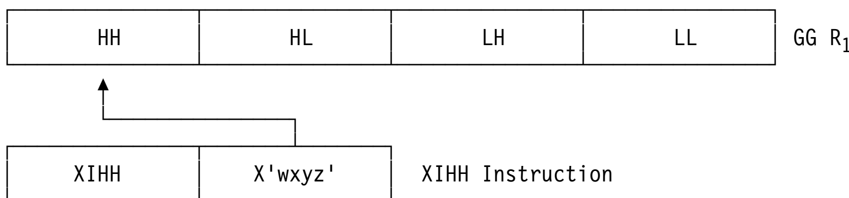
21.2.1. The AFI instruction with a negative operand does what the SFI instruction would do with an operand of the opposite sign. But: why is there a SLFI instruction?

21.2.2. This is the same solution as for Exercise 18.2.7, but with literal references replaced by immediate operands.

```
         LHI   2,10             * Count items in GR2
         SR    0,0                Accumulate sum in (GR0,GR1)
         LR    1,0                Initialize sum to zero
Loop     LH    4,0(,11)           Get an operand
         MH    4,0(,12)           Multiply by the other
         SRDA  4,32               Extend word product to 64 bits
         ALR   1,5                Add low-order words
         BC    12,NoCarry         Skip carry add if none to add
         ALFI  0,1              * Add a carry bit
NoCarry  ALR   0,4                Now add high-order words
         AHI   11,2             * Add 2 to address of first table
         AHI   12,2             * And to the address of second table
         AHI   2,-1             * Count down items by 1
         BP    Loop               Repeat if not done
         STM   0,1,DwSum          Store accumulated sum
```

Section 21.3

21.3.1. Consider the (possible) XIHH instruction: its 16-bit operand would be XORed with the high-order 16 bits of GG R_1, where X'wxyz' represents any arbitrary bit pattern.



Because XORing a 0-bit to any other bit leaves it unchanged,

```
         XIHH  reg,X'wxyz'        is equivalent to
         XIHF  reg,X'wxyz0000'    ... so XIHH is not needed.
```

21.3.2. Consider the NIHH instruction. Because ANDing a 1-bit to any other bit leaves that bit unchanged,

```
         NIHH  reg,X'wxyz'        is equivalent to
         NIHF  reg,X'wxyzFFFF'    ... so NIHH is not needed.
```

So, why are NIHH/HL/LH/LL present? System z implemented 16-bit immediate operands much earlier than support for 32-bit immediate operands, and RIL-type instructions like NIHF and NIHL appeared much later. But, the RI-type instructions also save 2 bytes in the instruction stream, in case your program needs to be as small as possible.

21.3.3. Consider OIHH: ORing a 0-bit to any other bit leaves the other bit unchanged, so that

```
        OIHH  reg,X'wxyz'        is equivalent to
        OIHF  reg,X'wxyz0000'    ... so OIHH is not needed.
```

See the last paragraph of the previous solution for details.

21.3.4. Use `NILH  1,X'FF00'`

21.3.5. Use `XIHF  7,X'80000000'`

21.3.6. You can use these three instructions:

```
        AHI   2,15               Force carry if not multiple of 16
        OILL  2,15               Set low-order 4 bits to 1
        XILL  2,15               Set low-order 4 bits to 0
```

21.3.7. The code will probably fail to give the intended result, because the NILL instruction operates only on the rightmost 16 bits of GR3. Thus, if GR3 had contained X'FFFFFFFF', after the NILL instruction c(GR3)=X'FFFF00F0', which is smaller than X'00000070' Similarly, if GR3 had contained X'12345678', its contents after the NILL instruction would be X'12340070', and the branch to **BitWas1** would be successful. These might not be the intended results.

21.3.8. These are possible critiques of the three sequences:

1. A word is required in storage for the literal.
2. Two instructions are required.
3. Three instructions are required, and the contents of GR5 is destroyed.

A better solution is

```
        NILF  4,X'3F'            Fullword immediate operand
```

21.3.9. This solution is almost correct (see the solution to Exercise 21.3.8). Because NILL uses a 16-bit immediate operand, it fails if there are any nonzero bits in the high-order half of GR4. Suppose c(GR4)=X'12345678': then

```
        NILL  4,X'003F'          yields c(GR4) = X'12340038'
```

which is probably not a satisfactory result.

Programming Problem 21.1.

This is another way to produce a hexadecimal addition table:

```
         Title 'Create a hexadecimal addition table'
         Print Nogen
P21_1    Start 0
         Using *,15               Establish addressability
         PrintLin HeadLine,L'HeadLine  Print heading line
         LHI   0,0                Initialize row value in GR0
         LHI   6,X'F'             Initialize digit mask
Col_Loop LHI   1,0                Initialize column value in GR1
         LR    2,0                Copy column value to GR2
         IC    2,Chars(2)         Get its EBCDIC representation
         STC   2,Line+1           Store in the print line
         LA    3,Line+3           Initialize product-value position
Row_Loop LR    5,1                Copy column value
         AR    5,0                Multiply by row value
         LR    4,5                Copy product to GR4
         SRL   4,4                Move to rightmost 4 bits
         NR    4,6                Leave high-order digit in GR4
         NR    5,6                Leave low-order digit in GR5
         IC    4,Chars(4)         Get EBCDIC char for high digit
         IC    5,Chars(5)         And for low digit
         STC   4,0(,3)            Store high digit in print line
         STC   5,1(,3)            Store low digit in print line
         AHI   3,3                Step to next print-line position
         AHI   1,1                Increment column value
         CHI   1,16               Check for finished with this line
         JL    Row_Loop           If not done, repeat for next column
         PrintLin Line,L'Line     Print a line of the table
         AHI   0,1                Step to next row
         CHI   0,16               Check if all rows done
```

```
         JL    Col_Loop           If not done, repeat for next row
         Printout *,HEADER=NO     Terminate the program
Chars    DC    C'0123456789ABCDEF'     EBCDIC characters
HeadLine DC    C'1   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F'
Line     DC    CL(L'Headline)' '
         End   P21_1
```

Section 22 Solutions

Section 22.1

22.1.1. Unfortunately, the letter "L" is used in many mnemonics to mean "Low" as well as "Long". The branch relative on condition with a 16-bit I_2 operand and mask 4 already uses the JL mnemonic, so the unconditional long relative branch needed a different name.

22.1.2. If your program uses no "program base" registers for instructions, it's possible that an instruction like

```
         NOP   X
```

could require base-displacement resolution. Using JNOP and JLNOP means you won't need a base register to resolve the operand address. (You could always write your NOP instruction using explicit base and displacement, but why do the extra work?)

Sometimes the operands of NOPs are used to indicate a position in the program. For example, if you wanted to "mark" a set of instructions that can be viewed by debuggers and other diagnostics, you could write

```
         JNOP  137               Marker for item 137's position
```

22.1.3. The generated machine instructions are:

```
(1) A7F4 FFFB
(2) A7F4 03FF
(3) A7F4 FFD8
(4) C0F4 0000 0001
```

Executing the last instruction will cause an operation exception, because it will branch into the second halfword of the instruction that contains zeros.

Section 22.2

22.2.1. The result will be the same as if you had written

```
Str      DC    CL80'String to be Scanned For Special Characters  '
```

22.2.2. The only changes needed are to these three statements:

```
         JNL   Okay              Branch if a letter or digit
         - - -
         CHI   1,80              Compare count to 80 (string length)
         - - -
         JL    GetChar           Loop if less than 80 done so far
```

Section 22.3

22.3.2. The indexed branch instruction labeled **A2** cannot be changed, because relative branch instructions cannot be indexed. However, you can fix this by rewriting it as

```
A2       LA    1,BrTbl(1)        Form address of target instruction
         BR    1                 Branch to it
```

If local addressability for the LA instruction is not available, you could use this approach:

```
A2       LARL  0,BrTbl           Address of start of branch table
         AR    1,0               Add to offset into the table
         BR    1                 Branch to it
```

Section 22.4

22.4.1. Replace the two instructions

```
Loop     LA    2,0(1,1)          (Count + Count) in GR2
         BCTR  2,0               (2 * Count) - 1
```

with the single instruction

```
Loop     LAY   2,-1(1,1)         (Count+Count) - 1 in GR2
```

22.4.2. Fixed-point overflows never occur in executing Branch on Count instructions.

22.4.4. A direct solution tests *every* bit, counting the one-bits, until the K-th bit is found. This solution uses some arithmetic-immediate instructions.

```
         SR    0,0                 Initialize bit offset
         L     1,KK                Initialize 1-bit count
         LA    2,Str               Initialize byte pointer
Outer    IC    3,0(,2)             Pick up a byte for testing
         SLL   3,24                Position at left end
         LA    4,8                 Initialize shift count
Inner    LTR   3,3                 Is the bit a 1-bit?
         JNM   ZeroBit             Branch if not
         AHI   1,-1                Have a 1-bit, deduct 1 from K
         JZ    Found               Finished if it's the K-th one
ZeroBit  AHI   0,1                 Increment bit offset
         SLL   3,1                 Out with the old, in with the new
         JCT   4,Inner             Count bits in this byte and loop
         AHI   2,1                 Increment byte pointer
         J     Outer               ... and go get a new byte
Found    ST    0,BitOff            Store the desired bit offset
```

22.4.5. Probably, something strange. If c(GRb) is even (as an instruction address must be) before executing the BCT, control will arrive at the branch location with an odd value (that address, minus 1) in GRb. If c(GRb) is +1, no branch occurs and the next instruction after the BCT will be executed. If c(GRb) is odd (and not +1), the program will try to branch to an odd address, but the number in GRb will then be even; a specification error will occur due to the odd branch address, which will be hard to find because the contents of the register will now be even. Don't do this!

22.4.6. The only thing needing care here is to be sure we store the first value.

```
         LA    1,100               Count list items in GR1
         LA    2,NewList-4         Output table addr in GR2 (delayed)
         LA    3,IntList           Input table addr in GR3
         L     0,IntList           Pick up first item
         J     Store               And enter in output list
Loop     L     0,0(,3)             Get item from list
         C     0,0(,2)             Compare to previous
         JE    Next                Don't store if equal
Store    LA    2,4(,2)             Increment storage address
         ST    0,0(,2)             Store into 'NewList' table
Next     LA    3,4(,3)             Increment 'IntList' address
         JCT   1,Loop              And repeat
         S     2,=A(NewList-4)     Subtract start address
         SRL   2,2                 Divide by element length
         ST    2,NumNews           Store number of new elements
```

22.4.7. It makes one extra iteration of the loop, and therefore calculates $(N+1)^2$.

22.4.8. The USING and BASR are reversed. At *execution* time, the “implied addresses” **AA** and **Loop** will actually refer to **AA+2** and **LOOP+2** instead! The sum in GR2 is therefore 2+3+4+5 instead of 1+2+3+4. (It may be worth reviewing Section 10.)

Here's the assembled program:

```
000000                      1 Ex22_4_8 START 0
                            2          USING *,8             Establish addressability
000000 0D80                 3          BASR  8,0             Set base register
000002 4140 0004            4          LA    4,4             Initialize counter
000006 4170 801C            5          LA    7,AA            Initialize address
00000A 1B22                 6          SR    2,2             Set sum box to zero
00000C 0700                 7 Loop     NOPR  0               Let the CPU catch its breath
00000E 4A20 7000            8          AH    2,0(,7)         Add a data item to the sum
000012 4170 7002            9          LA    7,2(0,7)        Increment address by 2
000016 4640 800C           10          BCT   4,Loop          Branch back if not done
00001A 0A03                11          SVC   3               Do something unforgettable
00001C 0001000200030004    12 AA       DC    H'1,2,3,4,5,6,7,8,9'   Table of numbers
000000                     13          END   Ex22_4_8
```

22.4.10. This solution uses the techniques of Section 19:

```
         XGR   0,0                  Set count in GG0 to zero
X        LTGR  2,1                  See if all bits of GG1 are zero
         JZ    Done                 If all zero, count is finished
         BCTGR 2,0                  Decrement GG2 by 1
         NGR   1,2                  Set rightmost bit of GG1 to zero
         AHI   0,1                  Increment count
         J     X                    Repeat until c(GG1) = 0
Done     - - -                      Count of 1-bits now in GG0
```

22.4.11. Consider these instructions:

```
         XR    1,1                  Set shift count to zero
Test     CLM   0,8,X'40'            Check for 1-bit in bit position 1
         JNL   Done                 Branch if it's there
         ALR   0,0                  Shift c(GR0) left one bit
         BCT   1,Test               Count down and loop
Done     LCR   1,1                  Make shift count positive
```

22.4.12. This shows the generated object code:

```
 Loc   Object Code        Assembler Language Statements
 0000  0DC0                        BASR  12,0
                                   Using *,12
 0002  4130 0040                   LA    3,64
 0006  4170 C03A                   LA    7,A
                                   Using A,7
 000A  5800 7000          Loop     L     0,A
 000E  5A00 7100                   A     0,B
 0012  5800 7200                   ST    0,C
 0016  4630 C008                   BCT   3,Loop
                                   Drop  7
 001A   <something>                PrintOut *
 003C                     A        DS    64F
 013C                     B        DS    64F
 023C                     C        DS    64F
```

22.4.13. The program actually calculates C(1)=A(1)+B(1) 64 times! To make it work correctly, insert

```
         LA    7,4(,7)
```

before the BCT instruction.

22.4.14. The errors are:

1. The SLL instruction should occur after testing the sign bit; otherwise only 31 bits will be tested.
2. The BZ instruction should be BNM; BZ will branch only if the entire contents of GR1 is zero.
3. An LA instruction can't be used to increment GR0.

The corrected code might look like this:

```
         SR    0,0                  Set count to zero
         LA    2,32                 Count 32 bits
Loop     LTR   1,1                  Test sign bit
         BNM   Next                 Branch if sign bit is zero
         A     0,=F'1'              Add 1 to count of 1-bits
         SLL   1,1                  Shift a bit into sign position
Next     BCT   2,Loop               Repeat for all 32 bits
```

22.4.15. This solution uses the RLL (Rotate Left Logical) instruction:

```
         SR    0,0                  Set count to zero
         LA    2,32                 Count 32 bits
Loop     LTR   1,1                  Test sign bit
         BNM   Next                 Branch if sign bit is zero
         AHI   0,1                  Add 1 to count of 1-bits
         RLL   1,1,1                Rotate a bit into sign position
Next     BCT   2,Loop               Repeat for all 32 bits
```

22.4.16. All five sequences yield the same result, the two's complement of the number in R0. The first two give the same (arithmetically valid) CC setting. The third will indicate CC values of either 2 (the result is zero), or 1 (the result is not zero). The fourth and fifth lead to CC settings of 0 (the result is zero) or 1 (the result is not zero).

22.4.17.

(1) The instructions add 1 arithmetically to c(GR0). If c(GR1)=X, the first LCR generates −X. The BCTR generates −X−1, and the final LCR generates −(−X−1) = X+1.
(2) They imitate the action of (a) AHI 0,1 or (b) A 0,=F'1' (among other possibilities).
(3) The Condition Code settings are identical for the two instructions noted in (2).[329]

Section 22.5

22.5.1. This solution uses indexed LH and STH instructions:

```
         LA    2,2                For incrementing/decrementing
         LH    1,NN               Start at end of HH array
         AR    1,1                N*2 (for halfword indexing of HH)
         SR    3,3                Index at RR starts at 0
LOOP     LH    0,HH-2(1)          Get word from high end of HH
         STH   0,RR(3)            Store at low end of RR
         AR    3,2                Increment RR index
         SR    1,2                Decrement HH index
         JP    LOOP               Repeat until index = 0
```

22.5.3. Do-Until.

Section 22.7

22.7.1. For BXLE, the branch will not be taken, and c(GR3) = 2. For BXH, the branch will be taken, and c(GR3) = 2 again.

22.7.2. For BXLE, c(GR3) = 6, and no branch occurs. For BXH, c(GR3) = X'C0000000'! The BXH instruction was executed 29 times.

22.7.3. The branch will occur only if the remainder was zero! If the remainder is greater than zero, the sum of remainder and quotient will be greater than the quotient in GR7, and no branch will occur.

22.7.5. The claim is true. If the sum c(GRx)+c(GRy) is non-negative, it is also $\leq 2^{31}-1$, so the claim holds. If the sum is less than zero it has overflowed, and the BXLE instruction does not branch. Logically, the sum cannot exceed $2 \times 2^{31}-1$, so subtracting $2^{31}-1$ gives the correct result modulo $2^{31}-1$.

Another way to say this: if a,b are $\leq$ m, and if a+b>m, then $a+b-m = a+b \pmod{m}$.[330]

22.7.6. Suppose the next-to-leftmost bit of the positive number is 1. Then the SLA will indicate an overflowed (not a negative) result. Then (1) the JM should be be JO or JNP, and (2) GR0 should be initialized to contain 1, not 0. The shift count in GR3 can be reduced to 31:

```
         LA    0,1                Initialize bit position
         LR    2,0                Increment of 1 in GR2
         LA    3,31               Comparand in GR3
X        SLA   1,1                Shift, test bit next to sign
         JO    Y                  Exit if it was 1
         JXLE  0,2,X              Count and loop if it was 0
Y        - - -
```

22.7.9. The BXLE instruction is executed 6 times, and branches 5 times. The sequence of values in GR1 is 7, 24, 41, 58, 75, and 92.

22.7.10. (C−A)/B+1 times. (If the value of the expression is not positive, the body of the loop is executed only once.)

329 A performance expert (Dan Greiner) tested all possible 2^{32} possible values to show their equivalence. The test took only a few seconds.

330 The “student” was Donald Knuth.

Section 22.8

22.8.1. The results are shown in this table:

c(GR1)	BXH	BXLE
0, 1	c(GR1)=0 (sum and comparand are 0)	Endless loop (sum and comparand are 0)
positive, ≥ 2	Endless loop (sum > comparand)	Set low-order bit of GR1 to 0 (sum > comparand)
negative	Set low-order bit of GR1 to 0 (sum −, comparand +)	Endless loop (sum −, comparand +)

Section 22.9

22.9.1. c(GR2) will be 40. The instructions calculate the largest doubling of c(GR2) not exceeding c(GR3). But be careful: If c(GR3)=X'7FFFFFFF' and c(GR2)=1, 2*c(GR2) will eventually overflow and become negative. At that point, after at most 32 iterations, c(GR2) will become zero, and the BXLE instruction will then loop endlessly. (Don't try it!)[331]

22.9.2. After the BXLE instruction has been executed once, c(GR5)=2. This is compared to the *original* value in GR5 (the comparand), which was 1. This is no longer less than or equal to the sum, so the final value in GR5 is 2.

For BXH, the sum is greater than the comparand each time BXH is executed, until the sum overflows after 31 iterations. c(GR5) then is negative, which when compared to the comparand X'40000000' is no longer greater, so the final value in GR5 is X'80000000'.

22.9.3. The two leftmost bits of the positive number in GR1 can be either 00 or 01. If they are 00, then the first BXH produces a larger *but still positive* number in GR1, and will branch to **ZBit** to increment the count. If the leftmost two bits are 01, the sum (formed by the BXH) in GR1 will become negative, and the BXH at **Loop** will not branch.

If the number in GR1 was arbitrary, a test for zero would be needed to exit the entire code sequence; if the number could be negative, LTR/JM instructions could indicate that bit number 0 was a 1-bit.

22.9.4. Consider the two high-order bits of GR1:

1. B'00' means that c(GR1) is positive, and the sum of the index and comparand will not overflow. The sum is greater than the comparand (the original c(GR1)), so the instruction will branch.
2. B'01' means that c(GR1) is positive, but the sum of the index and comparand will overflow and appear negative. The sum is then less than the comparand (the original c(GR1)), so the instruction will not branch.
3. B'10' means that c(GR1) is negative, but the sum of the index and comparand will overflow and appear positive. The sum is then greater than the comparand (the original c(GR1)), so the instruction will branch.
4. B'11' means that c(GR1) is negative, and the sum of the index and comparand will not overflow. The sum is then less than the comparand (the original c(GR1)), so the instruction will not branch.

22.9.5. Consider the two high-order bits of GR1:

1. B'00' means that c(GR1) is positive, and the sum of the index and comparand will not overflow. The sum is greater than the comparand (the original c(GR1)), so the instruction will not branch.
2. B'01' means that c(GR1) is positive, but the sum of the index and comparand will overflow and appear negative. The sum is then less than the comparand (the original c(GR1)), so the instruction will branch.
3. B'10' means that c(GR1) is negative, but the sum of the index and comparand will overflow and appear positive. The sum is then greater than the comparand (the original c(GR1)), so the instruction will not branch.
4. B'11' means that c(GR1) is negative, and the sum of the index and comparand will not overflow. The sum is then less than the comparand (the original c(GR1)), so the instruction will branch.

22.9.6. This exercise is most easily analyzed with a table:

331 I did, and that's why I know you shouldn't.

c(GR0)	c(GR1)	Branch?
0	any	No
>0	0	Yes
≤0	0	No
>0	>0, sum does not overflow	Yes
>0	>0, sum overflows	No
>0	<0	Yes
<0	<0, sum does not overflow	No
<0	<0, sum overflows	Yes
<0	>0	No

You can sometimes make efficient use of BXH and BXLE if the operands in the index and increment/comparand registers fit one of the patterns above.

22.9.7. The solution to this exercise uses the same table as in Exercise 22.9.6, except that each "Yes" and "No" is reversed.

22.9.8. If the R_1 operand is not equal to either the R_3 or $R_3|1$ operands, this description is correct. And since most uses of the branch on index instructions are used this way, little harm is usually done. However, consider these instructions:

```
LHI   6,2               c(GR6) = 2
LHI   7,2               c(GR7) = 2
BXLE  7,6,XXX           Will it branch?
```

In our description, the sum of the index in GR7 (2) and the increment in GR6 (2) is greater than the comparand in GR7 (2), so the branch will not occur.

In the incorrect description, the sum replaces the index before comparison: the sum (4) replaces the original index in GR7, which then becomes the comparand. Since those values are equal, a programmer could be misled into believing the branch *will* occur.

22.9.9. This is a form of what is known as the "shift-and-square" method for calculating integer powers. For example, suppose you must evaluate X**5.

- The first exponent bit tested is 1, so the code branches to `OneBit`, where the BXLE instruction finds that c(GR0) is not zero, so X is squared in GR5; this result is called the "work value".
- Then the next exponent bit (0) is shifted into GR1; the BXLE finds that it's zero, and branches to `TestMore`. (Remember that the BXLE instruction also leaves zero in GR1.) Since there is still a 1-bit in GR0, the BXH instruction branches to "Square", where the work value is squared, leaving X**4 in GR5.
- The last 1-bit of the exponent is shifted into GR1, and the BXLE test does not branch, so the work value is multiplied by c(GR3)=X, giving X**5.
- The final test at `TestMore` finds that c(GR0) is now zero, and control passes to `Finished`, leaving X**5 in GR3.

Programming Problem 22.1.

We will show two solutions to this problem. Both use similar techniques to generate the table; the first solution assumes fixed positions for all the data in the table.

```
         Title 'Solution to Problem 22.1'
P22_1    CSect ,        Print a hexadecimal multiplication table
         Using *,15              Establish base register
         Print NoGen
         PrintLin  Head,L'Head   Print heading line
         XR    0,0               Initialize row counter
Row_Loop DC    0H
         LR    2,0               Get row value
         IC    2,Chars(2)        Get character for this row
         STC   2,Line+1          Store at left end of print line
         XR    1,1               Initialize column counter
         LA    4,Line+3          Point to first table position
Col_Loop DC    0H
         LR    3,0               Put row value in GR3
         MR    2,1               Multiply by column value
         LR    2,3               Copy product to GR2
         SRL   2,4               Position high-order product digit
         NILL  3,X'F'            Low-order product digit in GR3
         IC    2,Chars(2)        Get equivalent EBCDIC character
         STC   2,0(,4)           Store high-order digit in print line
         IC    3,Chars(3)        Get equivalent EBCDIC character
         STC   3,1(,4)           Store low-order digit in print line
         AHI   4,3               Step to next print line position
         AHI   1,1               Increment column counter
         CHI   1,16              See if this row is complete
         JL    Col_Loop          Branch if not
         PrintLin Line,L'Line    Print this row
         AHI   0,1               Increment row counter
         CHI   0,16              See if all rows are complete
         JL    Row_Loop          If not, do another row
         PrintOut *,Header=NO    Terminate the program
Chars    DC    C'0123456789ABCDEF'
Head     DC    C'1   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F'
Line     DC    CL(L'Head)' '     Print line
         End
```

In this second solution, the symbols **Margin** and **Space** defined by EQU statements at the beginning of the program control the position of the left margin and the spacing between columns of the table. Otherwise, the program is straightforward.

```
         Title 'Alternate solution to Problem 22.1'
P22_1A   START 0
Margin   Equ   15                 Indentation from left edge
Space    Equ   5                  Space between columns
         BASR  15,0               Establish run-time base
         Using *,15               Let assembler know about assumption
         PrintLin Title           Print title at the top of the page
         LA    1,15               Set up 15 X values on the top line
         SR    2,2                Initialize pickup index to zero,
         LR    3,2                And the character storage index
TopRow   IC    0,Char+1(2)        Pick up an EBCDIC character
         STC   0,Line+Margin+Space+1(3) Store in line
         LA    2,1(0,2)           Increment X value by 1
         LA    3,Space(0,3)       Move store index over to next slot
         JCT   1,TopRow           Do 15 digits in all
         PrintLin Line            Print the top row of the table
*                                 c(GR1) = 0, c(GR2) = 15
*
NextRow  LA    4,1                Initial X value for each row
         LA    1,1(0,1)           Increment Y value by 1
         SR    3,3                Reset line store index
         IC    0,Char(1)          Fetch left-column character
         STC   0,Line+Margin      Store in line in left column
         LA    0,15               Set column count in GR0 for loop
*
NextCol  LR    7,1                Y value
         MR    6,4                * Y value
```

```
         SLDL  6,28                Place high-order digit in GR6
         SRL   7,28                Move low-order digit to end of GR7
         IC    6,Char(6)           Get appropriate ...
         IC    7,Char(7)                ...EBCDIC characters, and
         STC   6,Line+Margin+Space(3)   ...store first in line
         STC   7,Line+Margin+Space+1(3) ...and second
         LA    3,Space(0,3)        Increment line index
         LA    4,1(0,4)            Increment X value
         JCT   0,NextCol           Loop until line is complete
         PrintLin Line             Print the line
         JCT   2,NextRow           And loop until 15 rows are done
*
         PrintLin Title,1          Skip a page to get a clean result
         PrintOut *,Header=NO      Stop
*
Char     DC    C'0123456789ABCDEF' Ebcdic form of hex digits
Title    DC    CL121'1Hexadecimal Multiplication Table'
Line     DC    CL121'0'            Print line, with double spacing
         END   P22_1A
```

Section 23 Solutions

Section 23.2

23.2.1. Only the first is valid; the I_2 operands of the other two require more than 8 bits, so the Assembler will indicate an error.

23.3.1. The table of EBCDIC character encodings in Table 13 on page 87 shows that C'a' has representation X'81' and C'A' has representation X'C1'. ORing the representation of C' ', or X'40', into C'a' produces X'C1'.

Section 23.3

23.3.2. `NI Flags,B'01111110'`

23.3.3. `OI Flags,B'10000001'`

Section 23.4

23.4.1. Consider the following:

```
         LA    1,Data-1           Set pointer to left of string
         LR    0,1                Save start address for later
Loop     LA    1,1(0,1)           Point to next byte
         CLI   0(1),X'FF'         Check for all 1-bits
         JNE   Loop               Branch if not all ones
         SR    1,0                Subtract start addr to get count
```

23.4.2. No, because SI-type instructions can't be indexed!

23.4.3. This solution scans characters from right to left.

```
         LHI   0,80               Record length
         LA    1,Record+80-1      Point GR1 to last character
Test     CLI   0(1),C' '          Check for blank character
         JNE   Store              Branch if it's not a blank
         BCTR  1,0                Move pointer left by 1 byte
         JCT   0,Test             Count down and repeat
Store    ST    0,DataLen          Store data length
         ST    1,LastChAd         Store address of last nonblank
```

If there no nonblank characters, the address stored at **LastChAd** will be A(Record-1).

23.4.4. Try CLI *,X'FF' — the opcode for CLI is X'95', which is less than X'FF'.*

23.4.5. Try CLI *,0 — the opcode for CLI is X'95', which is greater than 0.*

23.4.6. He can write the test in either of two ways:

```
1.         CLI   Char,C'f'

2. LowerF  Equ   C'f'
           CLI   Char,LowerF
```

23.4.7. Try CLI *+1,0 — the CLI compares its immediate operand to itself.* A better instruction is CLR 0,0 which makes no memory reference, but does refer to a register.

Section 23.5

23.5.1. This is essentially the definition of a 9-bit two's complement binary integer, which has values between -2^8 and $+2^8-1$.

23.5.2. In step 1, if the I_2 mask AND the first-operand byte is zero, all the tested bits were zero. In step 2, some or all bits from the first step must have been ones. After XORing this field with the I_2 mask again, if all the remaining bits from step 1 were ones, the result is now zero, meaning all tested bits were ones. In step 3, if the result of the XOR is not zero, the tested bits were mixed zeros and ones.

* This may not be the best technique, because it mixes a reference to an instruction and data.

23.5.3. Try `TM *+1,AnyNonZeroValue` — because the I_2 mask has at least one nonzero bit, each one-bit in the mask will test itself, so all tested bits will be one.

23.5.4. TUM is not the mnemonic for Test Under Mask; the second operand is written as a decimal term (with value X'50'), and the branch mnemonic for a zero sign bit is BZ; BP would never branch, because TM doesn't set CC=2. He should have written:

```
         TM    BIN,X'80'          Test for zero sign bit
         BZ    POS                Branch if nonnegative
```

23.5.5. You can use `TM *+1,0`. (Is there any limitation on the first operand address?)

23.5.6. Yes, because BNM has mask B'1011'. But be careful! There are other TM-like instructions that don't behave like TM!

Section 23.6

23.6.1. There are many ways to do this; this is a representative solution:

```
         LHI   0,8                Initialize bit count
         LA    1,BitChars         Point to output character string
         ICM   3,B'1000',BitData  Source byte in leftmost byte of GR3
Repeat   SR    2,2                Clear GR2 for shift
         SLDL  2,1                Move a bit into GR2
         AHI   2,C'0'             Form ECBDIC 0 or 1 character
         STC   2,0(,1)            Store character in output area
         AHI   1,1                Increment output address
         JCT   0,Repeat           Repeat until 8 bits are done
```

23.6.2. No, because it doesn't correctly handle the mixed case.

Section 23.7

23.7.1. Yes. After the 'DS B' the Location Counter has been advanced by 1.

23.7.2. Consider these instructions:

```
(1)      NI    BitA,255-L'BitA-L'BitB  Set both bits to zero

(2)      XI    BitB,L'BitB        Invert BitB

(3)      TM    BitA,L'BitA+L'BitB Test both bits
         JO    Both               Branch if both are ones

(4)      LA    0,2                Assume both bits are ones
         TM    BitA,L'BitA+L'BitB Test the two bits
         JO    Done               Finished if both are ones
         BCTR  0,0                Reduce the ones-count by 1
         JM    Done               Finished if only one is one
         BCTR  0,0                Both 0, reduce ones-count to zero
Done     - - -
```

Why can't you use `AHI 0,-1` instead of `BCTR 0,0`? Because `AHI` sets the Condition Code, and the `JM` instruction will branch incorrectly.

Section 23.9

23.9.1. The number in GR6 is alternately 2 and 4, because the SLL instruction is alternately SLL and SRL. (To understand this, you'll have to revisit the opcodes shown in Table 78 on page 242!) The values in GR4 are therefore the odd numbers which are not multiples of 3.[332]

23.9.2. The OI and XI have zeros in the right hex digit of the immediate operand, and the NI has all ones.

23.9.3. It saves allocating a byte for `Flag`! But it also makes the program self-modifying. (At the time HASP was written, self-modifying programs weren't unusual.)

332 An interesting bit of code from a prime-number generator by Valdo Androsciani.

Section 24 Solutions

Section 24.2

24.2.1. None. The two words mean the same thing.

Section 24.3

24.3.1. No.

24.3.2. Yes and No. This may seem puzzling, because the assembler can derive an implied length attribute of an operand expression. But consider these statements:

```
A        LA    0,L'*              c(GR0) = 4
B        LA    0,L'*+3            c(GR0) = 7
C        LA    0,L'3+*            Error, L' operand not a symbol
D        LA    0,L'B              c(GR0) = 4
E        LA    0,8+L'B            c(GR0) = 12
F        LA    0,L'(8+L'B)        Error, L' operand not a symbol
```

The operand of an *explicit* Length Attribute Reference must be either a valid symbol or a Location Counter Reference. In the third and sixth examples, the expressions are 3+* and (8+L'B). Since the Length Attribute of an expression is that of its leftmost term, the Assembler issues an ASMA147E error message.

Section 24.4

24.4.1. (1) 2 (the length attribute of A is implicitly 2 bytes), X'0102'; (2) 4 (the length attribute of B is explicitly 4), C'ABCD'; (3) 3 (the Length Expression is specified explicitly), X'000000'!

24.4.2. The possible operand formats are shown in this table:

Format	As first operand	As second operand
7(4)	S(N)	D(B)
24(6,12)	D(N,B)	Invalid
A(B)	S(N) if B is absolute, otherwise invalid	D(B) if A and B are absolute, otherwise invalid
5(,1)	D(,B)	Invalid

24.4.3. Suppose the comma is omitted, so that the operand looks like this:

```
MVC   D1(B1),S2
```

Because this first operand of the MVC instruction was intended to specify an explicit address, it should have the third operand format described in Section 8.5. As written, the D_1 expression will be interpreted as an *implied* address, and the B_1 expression will be interpreted as the Length Expression.

Section 24.6

24.6.1. The result would be C'54345'. You might want to experiment with the CPU you are using to see what happens when you use MVCIN to move various strings onto themselves.

24.6.2.

```
MVC   Prefix,PText            Move prefix string
MVCIN Insert,IText+L'Insert-1 Move insert string
MVC   Suffix,SText            Move suffix string
```

24.6.3. C'DataData'.

24.6.4. The result fields are:

```
Result2  DS    C'PQRS'            From the first 4 bytes at Data2
Data2    DC    C'TUVWTUVW'
```

24.6.5. The effective address of the STC is the same in both cases. However, suppose you needed later to insert an instruction or two *between* the STC and the MVC: the *+5 operand would then refer to the wrong instruction. This shows why references between instructions should use symbols, not expressions involving the Location Counter. (But you should use an EX instruction, anyway!)

24.6.6. Sketching the operations may help. The first MVCIN moves the last byte of the second operand, C'G', to the first byte of the first operand, etc. The final byte of the second operand, C'A', is moved to the last byte of the first operand, which is at the same address!

A similar analysis holds for the second MVCIN, except that the initial byte moved from the second operand is at the same address as the first byte of the first operand. Thus, the one-byte overlap is harmless in both cases.

The result at X is C'GFEDCBA', and the result at Q is C'54321'.

24.6.7. A reviewer commented "No one in their right mind would/should do such a thing; would it work?"

```
         MVC   A-A+C,A
         MVC   B-B+L'A+C,B
```

Yes, it works!

24.6.10. For the instruction to work:

1. The symbol N must be absolute.
2. The term `L'Rec` must be the true length of each record.
3. Their product must be less than or equal to 256.

The instruction will not work if any of the above conditions is not true.

1. The value of the symbol N might be given by a value in memory.
2. `L'Rec` might not define the length of a record. For example, suppose the record had been defined by statements like these:

```
Rec      DS    0X                 Define start of record
Field1   DS    CL40               First field in the record
Field2   DS    ...                Similarly for other fields
RecLen   Equ   *-Rec              True length of the record
```

In this case, the value of `L'Rec` will be 1.

24.6.11. The instructions work correctly.

Section 24.7

24.7.1.

```
         MVC   Temp,ZZ            Move to work area
         NC    Temp,=X'00780000'  Isolate second integer
         NC    XX,=X'FF87FFFF'    Set new-integer position to zeros
         OC    XX,Temp            Insert new second integer value
         - - -
Temp     DS    XL4                Temporary work area
```

24.7.3. The instructions work correctly. Just before executing the NC instruction, the eight bytes at **`BitChars`** will contain

```
         X'9148241980040201'
```

The NC instruction will change the bytes to

```
         X'0100000100000001'
```

and the OC instruction inserts the high-order `X'F0'` bits to form the eight EBCDIC characters `C'10010001'`.

Section 24.8

24.8.1. Consider $-1 > -2$. Because CLC does unsigned byte comparisons, `X'FFFFFFFF'>X'FFFFFFFE'`, and the assertion is true.

24.8.2. Because negative binary integers have a 1 in the sign bit, a logical comparison will consider all negative numbers to be greater than all non-negative numbers.

Now, suppose you *invert* the sign bits: now, a logical comparison will consider all non-negative values to be greater than the originally negative values. For example:

arithmetically smallest	−20 = X'FFFFFFEC'	becomes	X'7FFFFFEC'	logically smallest
	−10 = X'FFFFFFF6'	becomes	X'7FFFFFF6'	
	+10 = X'0000000A'	becomes	X'8000000A'	
arithmetically largest	+20 = X'00000014'	becomes	X'80000014'	logically largest

So, these instructions will compare +10 to −10 correctly:

```
         XI    M10,X'80'          Invert sign of -10
         XI    P10,X'80'          Invert sign of +10
         CLC   P10,M10            Compare (modified) +10 to -10
         JH    P10Big             Branch if +10 > -10
         JL    M10Big             Branch if +10 < -10
         J     Equal              Branch if they're equal
         - - -
M10      DC    F'-10'
P10      DC    F'+10'
```

Comparisons using this technique should reset all inverted signs to their correct values.

24.8.3. It is not true for (3). The CC would be set to 0 if all the bytes were identical, whether or not they were zero.

24.8.4. Since only a single length is provided in the CLC instruction, there is no way to determine a “shorter operand”. The maximum number of bytes compared is always the number specified; if an inequality occurs, the comparison stops immediately. Also, there is no reference in the CLC instruction to “blanks” or “padding”. (We'll see padding in Section 25.)

24.8.5. This solution uses the “one byte at a time” property of the CLC instruction:

```
         CLI   Chars,C' '         Check first character for blank
         JNE   NotBlank           If not blank, don't bother the rest
         CLC   Chars+1(71),Chars  Compare the rest of the characters
         JE    AllBlank           Jump if all were blanks
NotBlank - - -                    Not all 72 characters are blanks
```

24.8.6. Because the length attribute of the literal `=120C' '` is 1, only a single byte will be compared.

24.8.7. The instruction will set CC=0 if all 8 bytes of the field addressed by GR4 are identical. (So the instruction is very useful.)

Section 24.9

24.9.1. Here is an Assembler listing showing the generated data:

```
000000 4040404040404040           1 TRTAble  DC    (C'a')C' '        Anything less than C'a' is blanked
000081 8182838485868788           2          DC    C'abcdefghi'      Letters are unchanged
00008A 40404040404040             3          DC    7C' '             Non-printing characters are blanked
000091 9192939495969798           4          DC    C'jklmnopqr'      Print letters as is
00009A 4040404040404040           5          DC    CL8' '            More non-printing characters
0000A2 A2A3A4A5A6A7A8A9           6          DC    C'stuvwxyz'       Last of the lower-case letters
0000AA 4040404040404040           7          DC    23C' '            Blank anything between 'z' and 'A'
0000C1 C1C2C3C4C5C6C7C8           8          DC    C'ABCDEFGHI'      Letters are unchanged
0000CA 40404040404040             9          DC    7C' '             Non-printing characters are blanked
0000D1 D1D2D3D4D5D6D7D8          10          DC    C'JKLMNOPQR'      Print letters as is
0000DA 4040404040404040          11          DC    CL8' '            More non-printing characters
0000E2 E2E3E4E5E6E7E8E9          12          DC    C'STUVWXYZ'       Last of the upper-case letters
0000EA 404040404040              13          DC    6C' '             Blank anything between 'Z' and '0'
0000F0 F0F1F2F3F4F5F6F7          14          DC    C'0123456789'     Digits print okay
0000FA 404040404040              15          DC    6C' '             Tail-enders are blanked too
                                 16          End
```

If you want to see all 256 bytes of generated data, add a

```
         Print Data
```

assembler instruction statement at the start of your program.

24.9.2. Only a single TR instruction is needed. The translate table replaces each byte with its rotated equivalent; it begins like this:

```
RotaTabl DC    X'0080018102820383048405850686078708809...'
* Argument:      000102030405060708090A0B0C0D0E0F101112...'
```

24.9.3. This translation table uses no hand-counted duplication factors:

```
TRTable  DC    (C'a')C' '          Anything less than C'a' is blanked
         DC    C'abcdefghi'        Letters are unchanged
         DC    (C'j'-C'i'-1)C' '   Non-printing characters are blanked
         DC    C'jklmnopqr'        Print letters as is
         DC    (C's'-C'r'-1)C' '   More non-printing characters
         DC    C'stuvwxyz'         Last of the lower-case letters
         DC    (C'A'-C'z'-1)C' '   Blank anything between 'z' and 'A'
         DC    C'ABCDEFGHI'        Letters are unchanged
         DC    (C'J'-C'I'-1)C' '   Non-printing characters are blanked
         DC    C'JKLMNOPQR'        Print letters as is
         DC    (C'S'-C'R'-1)C' '   More non-printing characters
         DC    C'STUVWXYZ'         Last of the upper-case letters
         DC    (C'0'-C'Z'-1)C' '   Non-printing characters are blanked
         DC    C'0123456789'       Digits print okay
         DC    (X'100'-C'9'-1)C' ' Tail-enders are blanked too
```

24.9.4. First, define the following data areas and tables; their order is important:

```
Blank     DC   C' '               Moved by 0'S in Table
InputRec  DS   CL80               Moved by 1-80 in Table
A1Format  DS   0F
Table     DC   80A(((*+4-A1Format)/4)*256*256*256)   320-byte string
```

Note that the table at **A1Format** begins `X'01000000020000000300000004...'`. Then, the following instructions do the job:

```
         TR    A1Format(240),InputRec-1     "Unpack" 60 characters
         TR    A1Format+240(80),InputRec-1  "Unpack" last 20 characters
```

24.9.5. The table must be 256 bytes long, because any bit combination is possible. The table is constructed so that the function byte is the "hex transpose" of the argument byte that would access it. The table would start like this:

```
TRANS    DC    X'00,10,20,30,40,50,60,70,80,90,A0,B0,C0,D0,E0,F0'
         DC    X'01,11,21,31,... etc. ...'
         - - -
         DC    X'0F,1F, ...,                        ... DF,EF,FF'
```

24.9.7. Using a TR instruction is simpler than the solution to Exercise 17.4.6. The bytes in the translate table contain the bit counts:

```
BitNoTbl DC    AL1(0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,...)
```

24.9.8. First, define the translate table:

```
Gather   DS    0CL40
         DC    40AL1(4*(*-Gather))     40-byte string
```

Then the following instructions do the job.

```
         MVC   OutRec(40),Gather
         MVC   OutRec+40(40),Gather
         TR    OutRec(40),A1Format
         TR    OutRec+40(40),A1Format+160
```

The "translation" must be done in two steps, because we cannot define a full 80-byte table at Gather. (Why? Because the values stored in the bytes would exceed 255.)

24.9.9. This can be done in several ways; this is one approach.

```
ASMTable DC    (X'100')C' '            Initialize to 256 blanks
         ORG   ASMTable+C'.'           Position at .
         DC    C'.'
         ORG   ASMTable+C'('           Position at (
         DC    C'('
         ORG   ASMTable+C'+'           Position at +
         DC    C'+'
         ORG   ASMTable+C'&&'          Position at &
         DC    C'&&'
         ORG   ASMTable+C'$'           Position at $
         DC    C'$'
         - - -                         Etc., for other special chars
         ORG   ASMTable+C'='           Position at =
```

```
         DC    C'='
         ORG   ASMTable+C''''          Position at '
         DC    C''''
         ORG   ASMTable+C'a'           Position at a
         DC    C'abcdefghi'
         - - -
         ORG   ASMTable+C'A'           Position at A
         DC    C'ABCDEFGHI'
         - - -
         ORG   ASMTable+C'0'           Position at 0
         DC    C'0123456789'
         ORG   ,                       Set LC to end of table
ASMTbEnd Equ   *
```

24.9.10. This solution shows a complete program that does both the right and left shifts. As with many problems involving TR and TRT, the important part of the solution is in the translate tables. The table named **TLeft** translates every byte whose leftmost digit is `X'X'` into a byte `X'0X'` (shifting the left digit right), and the table named **Right** translates every byte whose rightmost digit is `X'Y'` into a byte `X'Y0'` (shifting the right digit left). The OC instructions then merge the translated strings.

```
XShift   CSect ,
         Print NoGen
         Using *,15
         J     Start                   Jump over data definitions
Old      DC    X'123456789ABCDEF0'     Sample data to shift
New      DS    XL(L'Old)               Space for shifted result
Temp     DS    XL(L'Old)               Temporary working storage
Start    MVC   Temp(L'Old),Old         Copy data to temp
         TR    Temp(L'Old),TLeft       Shift left digits to right
         MVC   New,Temp                Move shifted left digits
         MVC   Temp(L'Old),Old         Copy data to temp again
         TR    Temp(L'Old),TRight      Shift right digits to left
         OC    New+1(L'Old-1),Temp     OR right digits, with offset
         PrintOut New                  Print right-shifted result
*
         MVC   Temp,Old                Copy data to temp
         TR    Temp(L'Old),TRight      Shift right digits to left
         MVC   New,Temp                Move shifted right digits
         MVC   Temp(L'Old),Old         Copy data to temp again
         TR    Temp(L'Old),TLeft       Shift left digits right
         OC    New(L'Old-1),Temp+1     OR left digits, with offset
         PrintOut New,*,Header=NO      Print left-shifted result
TLeft    DC    16X'0',16X'1',16X'2',16X'3',16X'4',16X'5',16X'6',16X'7'
         DC    16X'8',16X'9',16X'A',16X'B',16X'C',16X'D',16X'E',16X'F'
TRight   DC    16AL1(0,16,32,48,64,80,96,112,128,144,160,176,192,208,22+
               4,240)
         End
```

24.9.12. The instruction at **Q1** adds C'0' to the hex digit; if it is numeric (between 0 and 9), this is its EBCDIC representation. The comparison at **Q2** tests to see if it is as large as 250 (or `C'0'+X'A'`); if so, the hex digit must have been A through F. The AHI instruction at **Q3** then corrects the representation; it might better have been written

```
Q3       AHI   8,C'A'-C'0'-X'A'   Map A-F back to correct values
```

24.9.14. The second byte can be translated with the table constructed in the solution to Exercise 24.9.5. The `X'F'` can be inserted most easily with an OI instruction, but you *could* use another TR instruction and a new table if you were feeling extravagant.

24.9.15. The result at **OddTable** will be `X'00000202040404'`

24.9.16. The program *does* work. The precautions are:

1. Be sure that `L'InString` truly represents the number of input bytes, and that it does not exceed 128. (What would need to be done if is greater than 128?)
2. Be sure that the length attribute of `OutStrng` is twice the number of input bytes, by defining `OutStrng` as

   ```
   OutStrng DS    CL(2*L'InString)
   ```

3. Assuming that the previous two conditions have been satisfied, write the TR instruction as

```
         TR    OutStrng(2*L'InString),=C'0123456789ABCDEF'
```

so that all operations depend only on the length of InString.

24.9.17. The required translate table can be created in several ways. First, you can do each byte "manually" so that the table starts like this:

```
RevBits  DC    X'008040C020A060E0109050D030B070F00804...'
```

but this is clearly a tedious and error-prone approach. A better approach is to let the Assembler do the work, but constructing the operand is not obvious.

We start with the observation that a bit representing the N-th power of 2 can be extracted from a byte named X using the expression

$$(X-((X/2^{(N+1)})*2^{(N+1)}))/2^{N}$$

To put this bit in the reversed position in the output byte, it must be multiplied by $2^{(7-N)}$. Using these expressions, we can create the translate table this way:

```
R        DC    256AL1((*-R)/128+2*(((*-R)-(((*-R)/128)*128))/64)+4*(((*X
               -R)-(((*-R)/64)*64))/32)+8*(((*-R)-(((*-R)/32)*32))/16)+X
               16*(((*-R)-(((*-R)/16)*16))/8)+32*(((*-R)-(((*-R)/8)*8))X
               /4)+64*(((*-R)-(((*-R)/4)*4))/2)+128*((*-R)-(((*-R)/2)*2X
               )))
```

Consider the second expression 2*(((*-R)-(((*-R)/128)*128))/64): this extracts the second bit of the offset from the start of the table, corresponding to value 2^6, and multiplies it by 2 so it will be the second bit from the right in the corresponding byte of the translate table.

24.9.19. After first time, the table will be X'00010203...7D7E7F 7F7E7D...03020100' (first half, then reversed) each time after the first. The original table is never recovered! (Can you explain this behavior?)

24.9.20. The table is unchanged every time the TR instruction is executed.

Section 24.10

24.10.2. You can't replace DC XL256'0' by DS because the other 254 bytes of the table won't be correctly initialized.

24.10.7. This complete program uses a translate-and-test table with nonzero entries for all nonblank characters.

```
X24_10_7 Csect ,
         Using *,15
         Print NoGen
RecLen   Equ   80                 Record length
         XR    1,1                Clear GR1
         XR    3,3                c(GR3) = worst-case string length
         LA    4,Record+Reclen-1  c(GR4) = A(last byte of Record)
         MVCIN Temp,0(4)          Move Record to Temp in reverse order
         TRT   Temp,FindNonB      Scan in reverse to find a nonblank
         JZ    Store              Branch if record was all blanks
         LA    3,Temp+Reclen      c(GR3) = 1 byte after Temp
         SR    3,1                c(GR3) = length of string
         LA    4,Record-1(3)      c(GR4) = A(last nonblank character)
Store    ST    3,DataLen          Store length of string
         ST    4,LastChAd         Store address of last nonblank
         Printout DataLen,LastChAd,*,Header=NO    Print results
DataLen  DS    F                  Significant length of string
LastChAd DS    A                  Address of last nonblank char
Record   DC    CL(RecLen)'This is a test to see what happens.'
Temp     DS    CL(RecLen)
FindNonB DC    256X'1'            Set all 256 bytes to X'1'
         Org   FindNonB+C' '      Position at offset X'40'
         DC    X'0'               We won't stop on blanks
         Org   ,                  Reset Location Counter
         End   X24_10_7
```

24.10.8. Suppose the second TRT does *not* find a nonzero function byte. Then the LTR 2,2 instruction would use the function byte from the first TRT, probably giving incorrect results.

24.10.9. The first operand generates 64 X'01' bytes, corresponding to offsets from X'00' to X'3F'. The second operand generates a single X'00' byte at offset X'40'. Finally, the third operand generates 256-C' '-1, or 256-64-1=191 bytes containing X'01'. Thus, a total of 256 bytes are generated.

24.10.11. The TRT scan stops at the X'02' function byte of XX, with Condition Code 1 meaning the scan is incomplete, and c(GR2)=X'00000002'.

24.10.14. The program will continue searching for a blank character somewhere prior to the start of the string, with unsatisfactory and possibly disastrous results.

Section 24.11

24.11.1. The two sets of USING statements are distinct and independent. The EX reference to the target instruction is resolved at the location of the EX instruction, while the operands of the target instruction are resolved with the USINGs in effect at that location. Be careful!

24.11.2. If EX “branched” to the target instruction, he would expect GR1 to contain the address of whatever is named **There**. Because EX does not branch, GR1 will actually contain the address of **Here**.

24.11.3. GR1 will contain the address of the LR instruction, *not* the address of the AR instruction! This is because the BASR places the Instruction Address into R_1; the PSW still contains the address of the instruction following the EX. The target instruction (the BASR) does not modify the IA in the PSW, because its R_2 digit is zero. The ILC will contain B'10' (indicating a 4-byte instruction), because EX is an RX-type instruction.

24.11.5. In this solution, we must make two checks: (1) to verify that the number of bytes to move is positive, and (2) test whether the number is greater than 256. If so, we move blocks of 256 bytes until the residual amount to move is ≤ 256 bytes.

```
         LTR   3,3               Check number of bytes to move
         JNP   Finished          If not positive, we're done
Test     CHI   3,256             Test if number to move is > 256
         JNH   LastGrp           If not, go handle residual group
         MVC   0(256,2),0(1)     Move 256 bytes
         AHI   1,256             Increment 'from' address
         AHI   2,256             Increment 'to' address
         AHI   3,-256            Decrement count by 256
         J     Test              And see of any more blocks to move
LastGrp  BCTR  3,0               Make it a machine length
         EX    3,MoveLast        Move the residual byte count
Finished - - -
         - - -
MoveLast MVC   0(*-*,2),0(1)     Executed MVC
```

Compare this to the instructions in Figure 220 on page 392, and explain the differences.

24.11.6. This solution handles pairing, but does not enforce the HLASM rule that character constants and self-defining terms must always have paired ampersands and apostrophes.

```
         LA    1,DCData          Initialize 'From' pointer
         LA    4,DCGen           Initialize 'To' pointer
         LR    3,4               Save 'To' start address for length
         LHI   2,L'DCData        Get 'From' string length
TestCh   CLI   0(1),C'&&'        Check for ampersand
         JE    HaveChar          Branch to check for pair
         CLI   0(1),C''''        Check for apostrophe
         JE    HaveChar          Branch to check for pair
CopyChar MVC   0(1,4),0(1)       Move 'From' character to 'To'
         AHI   1,1               Step 'From' pointer
         AHI   4,1               Step 'To' pointer
         JCT   2,TestCh          Test next character
         SLR   4,3               Final 'To' final addr - start addr
         ST    4,DCGenL          Store length of result
         J     Done
HaveChar CHI   2,1               Was it the last character?
         JE    CopyChar          Branch if yes, include it
         CLC   0(1,1),1(1)       Does next character match?
         JNE   CopyChar          Branch if not to keep it
         BCTR  2,0               Decrement count of remainder
         AHI   1,1               Step over the paired character
         J     TestCh            And check for more characters
Done     - - -
         - - -
DCData   DC    C'This ''string'' contains &&'s and ''s'
```

```
DCGen    DS    CL(L'DCData)       Result string
DCGenL   DS    F                  Length of result
```

24.11.8. The easiest way to do this is to “pre-pad” the receiving field if N > M.

```
         LHI   2,M                Length of source field
         LHI   3,N                Length of target field
         CR    2,3                See which is longer
         JE    Move               If the same, just move
         JH    Pad                If N>M, must pad target field
         LR    2,3                N<M, move only N characters
         J     Move               And move them now
Pad      MVI   DataPad,C' '       Initialize first pad byte
         LR    1,3                Copy N
         AHI   1,-2               Deduct for EX machine length
         JNP   Move               If not >0, only 1 pad byte
         EX    1,PadBlank         Otherwise complete the padding
Move     BCTR  2,0                Now have lesser of (N,M)-1
         EX    2,MoveData         Move the source bytes to target
         - - -
MoveData MVC   DataPad(*-*),Data       Move the data
PadBlank MVC   DataPad+1(*-*),DataPad  Ripple blanks
```

24.11.9. Suppose the two operands are byte strings named A and B, and we want to compare A to B.

```
         LHI   2,L'A              Length of A
         LHI   3,L'B              Length of B
         CR    2,3                Compare lengths
         JE    Compare            Equal, do the compare directly
         JH    ALonger            Branch if A is the longer string
BLonger  CLC   A,B                A is shorter, compare to B
         JNE   Done               If an inequality, we're dond
         LA    1,L'B-L'A          Calculate length of A pad
         LA    2,L'A+B            Where to start padded compare
APad     CLC   =C' ',0(2)         Compare tail of B to blank
         JNE   Done               If a mismatch, we're done
         LA    2,1(,2)            Step to next byte of B, CC unchanged
         JCT   1,APad             And compare again
         J     Done               Strings are equal, with padding
ALonger  CLC   B,A                B is shorter, compare to A
         JNE   Done               If an inequality, we're dond
         LA    1,L'A-L'B          Calculate length of B pad
         LA    2,L'B+A            Where to start padded compare
BPad     CLC   0(1,2),=C' '       Compare tail of A to blank
         JNE   Done               If a mismatch, we're done
         LA    2,1(,2)            Step to next byte of A, CC unchanged
         JCT   1,BPad             And compare again
Done     - - -                    CC set correctly for all compares
```

While it is possible at assembly time to determine which of L'A and L'B is larger and then define a string of blanks whose length is the absolute difference, the above approach seems simplest.

24.11.12. The key to this solution is the translation table, in which each function byte rotates the bits of the source byte to the right by one bit position.

```
         NILF  1,B'111'           Use only 3 rightmost bits of GR1
         JZ    Done               Finished if shift count was zero
Rotate1  TR    Rotator,RoTable    Rotate by 1 bit position
         JCT   1,Rotate1          Repeat until N shifts are done
Done     - - -
         - - -
RoTable  DC    X'00,80,01,81,02,82,03,83,04,84,05,85,06,86,07,87'
         DC    X'08,88,09,89, ... etc. ...                      '
         - - - etc.
         DC    X'              ... etc. ...                77,F7'
         DC    X'78,F8,79,F9,7A,FA,7B,FB,7C,FC,7D,FD,7E,FE,7F,FF'
```

You could avoid looping by choosing one of seven different “rotation” translate tables, but the gain seems less than the needed effort.

You can't rotate by N bits using a single translate table, because the source byte could contain any of 256 bit combinations, and the function bytes for each of 7 possible rotations are at different offsets in the table.*

24.11.13. Suppose the rightmost 8 bits of GR9 are B'00000011', indicating an odd number. Then, the executed TM instruction is effectively

```
         TM    OneBit,B'00000011'
```

But because only one of the tested bits at **OneBit** is one, the JO instruction would not branch to **OddReg** as required, because the tested bits are mixed zero and one.

24.11.14. This can be done by testing the N rightmost bits one at a time in a loop; assume GR0 can be used as a working register, and the number to be tested is in GR9.

```
         LA    0,N                 Number of bits to test
Test     EX    9,TMTest            Test the rightmost bit
         JNZ   NotPower            If 1, not a multiple of 2**N
         SRL   9,1                 Move to next significant bit
         JCT   0,Test              And try again
         - - -
NotPower - - -
         - - -
TMTest   TM    OneBit,0            Test the rightmost bit
OneBit   DC    B'00000001'         Low-order 1 bit
```

The limitations of this technique are:

1. GR0 cannot be the register tested.
2. If we aren't allowed to shift the contents of the register, we can't test for powers greater than 8.
3. If we can shift, the original contents of the register is destroyed, and there is no way to check the lost bits.

To do this more general test, if N is known at assembly time we could write

```
         L     0,PowerN            Get the test mask
         BCTR  0,0                 Decrement to form N 1-bits
         NR    0,9                 See if the bits of GR9 are zero
         JNZ   NotPower            Branch if not
         - - -
PowerN   DC    FS(N)'1'            Generate 2**N
```

24.11.15. The main problem is that the rightmost 4 bits of GR1 will modify the operation code of the executed LHI instruction! Thus, he should have written

```
         SLL  1,4                  Position register number correctly
         EX   1,LHIOp              Load the constant into a GPR
         - - -
LHIOp    LHI  0,137                Constant to be loaded somewhere
```

24.11.16. This solution uses EX and MVI instructions:

```
         LHI   0,8                 Set bit counter to 8
         LA    1,Char              Point GR1 to characters
         ICM   3,B'1000',Byte      Byte in high-order byte of GR3
A        XR    2,2                 Clear GR2
         SLDL  2,1                 Shift a bit from GR3 into GR2
         EX    2,MVI               Let the MVI "store" the character
         AHI   1,1                 Increment the address by 1
         JCT   0,A                 Count down by 1 and loop
         - - -
MVI      MVI   0(1),X'F0'          Store C'0' or C'1'
Byte     DC    X'CD'               Sample bit pattern
Char     DS    CL8
```

24.11.18. First, we create a translate table that can be used to reverse the order of a string of up to 256 bytes:

```
RevTbl   DC    256AL1(255-(*-RevTbl))
```

* In other words, I couldn't figure out how to do it. Can you?

This table contains 256 byte values starting at X'FF' and decreasing by 1 for each generated byte. Then, we can do the "inverse move" by first computing the position of the last L bytes of the table, moving them to the **Target** field, and then translate the source using these instructions:

```
         LR    1,0                Copy L to GR1
         BCTR  1,0                Make effective length in GR1
         LA    2,RevTbl+256       Point to byte past end of table
         SR    2,0                Point to last L bytes of table
         EX    1,Move2Tgt         Move those last L bytes to Target
         EX    1,Tran2Tgt         Translate from Source to Target
         - - -
Move2Tgt MVC   Target(*-*),0(2)   Moves L bytes to Target field
Tran2Tgt TR    Target(*-*),Source Moves L bytes in inverse order
```

This technique was often used before MVCIN was available.

24.11.19. Since the rightmost byte of GR1 is ORed into the second byte of the EX target instruction now in the IR, that second byte will be 9|5 for the first operand digit and 8|2 for the second operand digit, yielding SR 13,10 as the executed instruction.

24.11.20. EX is an RX-type instruction, so it can be indexed.

Programming Problem 24.2.

This solution uses three translate tables: one to do the shuffling, one to convert the hex digits to EBCDIC, and a third to place all the EBCDIC characters (including the shuffle count) into two output lines. (Note that this is the "Vertical hex" technique used in Problem 24.12!) The first output line has a blank carriage control character (single space), the shuffle count, a space, and then a blank followed by the two hex digits of a shuffled value. The second output line has the same format, except that the first four characters are blank.

Note that only the (even!) value of the symbol NC needs to be changed to modify the number of cards to shuffle. (The symbol MaxCount is defined as a safety check.)

```
         Print Nogen,Data
P24_2    CSECT ,
         USING *,15
NC       Equ   52                  Number of playing cards
MaxCount Equ   NC*2                Maximum number of shuffles(?)
*                                  Arbitrary upper limit
         XR    1,1                 Initialize shuffle counter
         XR    2,2                 Clear even work register
         XR    3,3                 Clear odd work register
NextOut  DC    0H                  Top of shuffling loop
         STC   1,Count             Store shuffle count
         LA    0,NC+1              Number of digits to format
         LA    4,Count             Addr(start of digit string)
         LA    5,Expand            Addr(start of expansion string)
ToHex    DC    0H                  Top of expansion loop
         IC    2,0(,4)             Get a byte
         SRDL  2,4                 Move low-order digit to GR3
         STC   2,0(,5)             Store high-order digit
         SRL   3,28                Position low-order digit
         STC   3,1(,5)             Store in next output position
         AHI   4,1                 Increment input pointer
         AHI   5,2                 Increment output pointer
         JCT   0,ToHex             Expand all the digits
         TR    Expand,Hex2Char     Convert X'0?' digits to EBCDIC
         MVC   Out,Format          Move formatting translate table
         TR    Out,Blank           Translate complete output lines
         PrintLin Out,L'Out/2      Print first set of shuffled values
         PrintLin Out+L'Out/2,L'Out/2   Print second set
         PrintLin Blank,1          Blank line
         AHI   1,1                 Increment count
         C     1,=A(MaxCount)      Test for enough shuffles
         JH    Finish              Exit if yes
         MVC   Temp,Shuffle        Copy shuffle table
         TR    Temp,Cards          Do the next shuffle
```

```
         MVC   Cards,Temp          Put the shuffled cards back
         J     NextOut             And go output the result
Finish   DC    0H
         PrintOut *,Header=NO      Terminate the program
*        Note: Count must immediately precede Cards
Count    DC    X'0'                Count of number of shuffles
Cards    DC    0XL(NC),(NC)AL1(*-Cards+1)   Starting card order
Temp     DS    XL(NC)              Shuffling area
*                                  Define the Shuffle table
Shuffle  DC    (NC/2)AL1((*-Shuffle)/2,(*-Shuffle)/2+NC/2)
*        Note: Blank must immediately precede Expand, for output
Blank    DC    C' '                Single space for carriage control
Expand   DS    XL(2*(NC+1))        Digit-expansion area
Out      DS    XL(2*((NC/2)*3+4))  Two formatted print lines
Format   DS    0XL(L'Out)          Length of formatting table
         DC    AL1(0,1,2,0)        Blank, 2 count digits, blank
F1       DS    0AL1                Anchor for first line
         DC    (NC/2)AL1(0,((*-F1)/3)*2+3,((*-F1)/3)*2+4)
         DC    4AL1(0)             4 blanks for second line
F2       DS    0AL1                Anchor for second line
         DC    (NC/2)AL1(0,((*-F2)/3)*2+NC+3,((*-F2)/3)*2+NC+4)
Hex2Char DC    C'0123456789ABCDEF' Hex-to-EBCDIC translation table
         End   P24_2
```

The output from this program showing the original order of the cards and the results after the first shuffle:

```
 00  01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A
     1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34

 01  01 1B 02 1C 03 1D 04 1E 05 1F 06 20 07 21 08 22 09 23 0A 24 0B 25 0C 26 0D 27
     0E 28 0F 29 10 2A 11 2B 12 2C 13 2D 14 2E 15 2F 16 30 17 31 18 32 19 33 1A 34
```

Some sample results with other numbers of cards:

Cards	Shuffles
8	3
10	6
12	10
24	11
52	8
64	6

Programming Problem 24.3.

The key to this solution is the translate table, which has zero function bytes in the positions of a blank and the decimal digits. There are better ways to create the (printable) column number!

```
         Print NoGen
P24_3    Start 0
         Using *,15              Use caller's GR15 as base register
Read     ReadCard Record,EndFile Read a record
         LA    0,Record-1
         LA    1,Record
         TRT   0(L'Record,1),Table     Scan the record
         JZ    Valid             No invalid characters found
         MVC   OutRec,Record     Move the record to the display line
*                                Invalid character found
         SR    1,0               Calculate column number
         SR    0,0               Clear high-order register
         D     0,=F'10'          Divide by 10
         MVC   InvalMsg,ErrorMsg Move error message text to line
         STC   1,ErrorCol        Store tens digit of column number
         STC   0,ErrorCol+1      Store low-order digit of column
         OC    ErrorCol(2),=2C'0' Add numeric zones
         PrintLin OutLine,OutLen Display the error message
```

```
         MVC   OutLine+1(OutLen-1),OutLine   Clear the display line
Valid    J     Read               Repeat for another record
EndFile  BR    14                 Return to the caller
Table    DC    (C' ')X'1',X'0'    Blank valid, lower characters not
         DC    (C'0'-C' '-1)X'1',10X'0' Numeric digits valid, rest not
         DC    (255-C'9')X'1'     Remaining characters invalid
TabLen   Equ   *-Table            Must be 256 (X'100')
ErrorMsg DC    C'Invalid character, column'
Record   DS    CL80               Input buffer for 80-byte records
*
OutLine  DC    C' '               Carriage control character
OutRec   DC    CL(L'Record)' '    Space for the record
         DC    C' '               Space
InvalMsg DC    CL(L'ErrorMsg)' '  Space for error message
         DC    C' '               Space
ErrorCol DC    2C' '              Space for column number
OutLen   Equ   *-OutLine          Length of the display line
         End   P24_3
```

Section 25 Solutions

Section 25.1

25.1.1. When base-displacement addresses are generated, a zero base or index digit is treated as "no base" or "no index". Since these instructions don't use base-displacement addressing, GR0 is allowed to contain an operand address.

25.1.2. Interruptible instructions like MVCL and CLCL are treated specially if they are interrupted when targets of an EX instruction. Instead of "backing up" the Instruction Address by 2 bytes (as indicated in Figures 222 and 225), the IA is reduced by 4 bytes so that the EX instruction will be re-executed when control returns to your program. Because the 4 operand registers will have been updated, be sure none of them is used as the R_1, X_2, or B_2 operand of the EX instruction.

25.1.3. No. Because only one source byte is involved, it is either

- not in the target area, or
- is the only source byte in the target area, in which case overlap occurs only if it is moved to itself, so the overlap isn't destructive!

25.1.4. It depends on the length of the second operand: if it is positive, the CC is set to 1 because the first operand was exhausted first; if the second operand length is also zero, the CC is set to 0.

25.1.5. In this instruction sequence, is the first XR instruction needed?

```
LA    0,Field+1          Target address
LHI   1,3500-1           Target length
XR    2,2                Source address
XR    3,3                Source length
ICM   3,B'1000',Field    Pad byte
MVCL  0,2                Propagate the byte
```

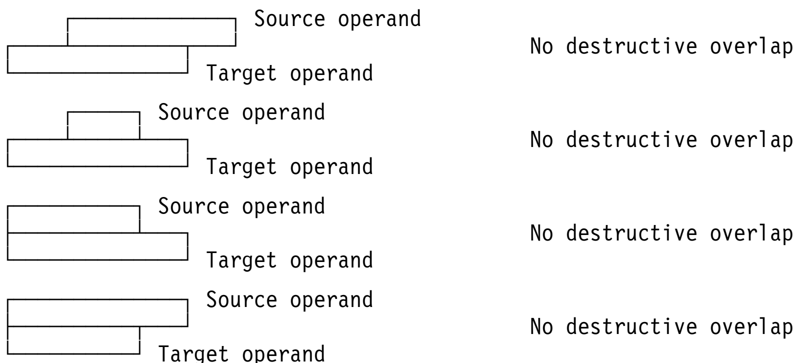
The `XR 2,2` isn't really needed, because the length of the source operand is zero, so no memory reference will be made to the "source" operand other than the pad byte.

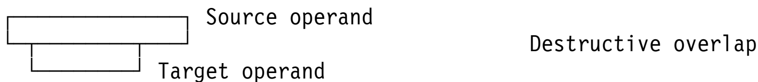
25.1.6. Yes. Even if the full operand lengths cause them to overlap, no target byte is used as a source byte. However, these tests don't indicate non-destructive overlap; see Exercise 25.1.9.

25.1.7. The results of the four operations are shown in this table; for case (2), remember that the length attribute of **STR2** is only 1, not 110!

Case	GR0, GR2 before	GR1, GR3 before	CC	GR0, GR2 after	GR1, GR3 after
(1)	X'074212D0' X'074212D4'	X'00000004' X'00000004'	1	X'074212D3' X'074212D7'	X'00000001' X'00000001'
(2)	X'074212D0' X'07421348'	X'00000078' X'00000001'	0	X'07421348' X'07421349'	X'00000000' X'00000000'
(3)	X'074212D0' X'074212E4'	X'00000014' X'00000002'	2	X'074212D4' X'074212E6'	X'00000012' X'00000000'
(4)	X'074212D0' X'074212D0'	X'00000005' X'00000004'	0	X'074212D5' X'074212D4'	X'00000000' X'00000000'

25.1.8. Some typical cases are shown:





How are these sketches affected by the presence of the pad byte?

25.1.9. First, we must analyze the possible overlap conditions. c(GR1) is the address of the leftmost byte of the second (Source) operand, and c(GR2) is the address of the leftmost byte of the first (Target) operand. First, we test for possible overlap conditions:

- If A(Source)<A(Target) *and* A(Source)+Length>A(Target), the overlap is destructive.
- If A(Target)+Length≤ A(Source), there is no overlap because the target area lies entirely at lower addresses than the source.
- If A(Source)+Length≤ A(Target), there is no overlap because the source area lies entirely at lower addresses than the target.
- If A(Source)=A(Target), the overlap is perfect, and is not destructive.

The symbols naming the registers help clarify the instructions.

```
RS       Equ   1                 Source operand address
RT       Equ   2                 Target operand address
RL       Equ   3                 Operand length
         CR    RS,RT             Check source vs. target addresses
         JNL   NoDest            Not destructive if source is higher
         LR    0,RS              Copy source address
         AR    0,RL              Add operand length
         CR    0,RT              Compare to target address
         JH    Destroy           Operands overlap destructively
NoDest   MVI   OLapFlag,0        Test for any overlap: Set flag off
         CR    RS,RT             See if the overlap is perfect
         JE    SetOlap           Set overlap indicator
         LR    0,RT              Copy target address
         AR    0,RL              Add operand length
         CR    0,RS              Compare to source address
         JNH   NoOlap            Jump if target lies below source
         LR    0,RS              Copy source address
         AR    0,RL              Add operand length
         CR    0,RT              Compare to target address
         JNH   NoOlap            Jump if source lies below target
SetOlap  MVI   OlapFlag,1        Indicate overlap occurs
NoOlap   CHI   RL,256            Test remaining length to move
         JNH   DoLast            If 256 or less, do final fragment
         MVC   0(256,RT),0(RS)   Move 256 bytes
         AHI   RS,256            Increment source address
         AHI   RT,256            Increment target address
         AHI   RL,-256           Decrease length remaining to move
         J     NoOlap            And test for more to do
MoveLast MVC   0(*-*,RT),0(RS)   Move remaining group of bytes
DoLast   BCTR  RL,0              Decrease last count for EX
         EX    RL,MoveLast       Move the last set of bytes
         TM    OlapFlag,1        Test if overlap indicator was set
         JO    Overlap           Branch if it was
         - - -
OlapFlag DC    X'0'              Byte for the overlap flag
```

25.1.12. See Figure 221 on page 404!

25.1.13. No, because there is destructive overlap. For example:

```
         LA    0,X+1             Target address
         LA    1,5               Target length
         LA    2,X               Source address
         LA    3,4               Source length
         ICM   3,8,=C'0'         Padding character
         MVCL  0,2               Move, starting at X, to X+1
         - - -
X        DS    CL5
```

results in Condition Code 3.

25.1.14. Consider these instructions:

```
         SR    1,1                Set source length and pad to zero
         LA    2,New              Target address
         LHI   3,8192             Set target length
         MVCL  2,0                Zero the field at New
```

Did you try to initialize GR3 using an LA instruction?

Section 25.2

25.2.1. If the instruction is completed with Condition Code 3 and is re-executed, the padding character might change before the entire compare or move operation is complete.

25.2.2. No; only the shorter operand is padded. If the operand lengths are equal, no padding is needed.

25.2.3. First, consult the *z/Architecture Principles of Operation* to see what it says about "address wrap-around". Then, if you decide to try executing the MVCLE in 24-bit addressing mode with a very long length, don't tell anyone where you got the idea.

25.2.4. `X'2625A000'`.

Section 25.3

25.3.1. Consider these three statements: although they define the required C-strings, first ask what are the length attributes of the three symbols?

```
Len0     DC    X'0'               C-string with length zero
Len1     DC    C'*',X'0'          C-string with length one
Len10    DC    10C'*',X'0'        C-string with length ten
```

The length attributes are 1, 1, and 1. You might have written

```
Len1     DC    X'5C00'            C-string with length one
```

but this technique is very clumsy for longer strings, and you must know the representation of each character. Be careful when you use length attributes with C-strings!

Section 25.4

25.4.1. You can do this by remembering the address of the most recent blank character before the terminating null byte is found. Another easy way to do this is to use SRST to find the first null, indicating the end of the C-string. Then, use TRTR to scan backward for the last nonblank character. A test for a null C-string is included in this example.

```
         LA    2,CData            GR2 has address of start of string
         LR    5,2                Copy starting address to GR5
         AL    5,CDataLen         GR5 points past last possible byte
         XR    0,0                Search character is a null byte
FindEnd  SRST  5,2                Scan for the terminating null byte
         JO    FindEnd            Repeat if not found yet
         JH    BadData            Error! No terminating null found!
         CR    5,2                Check whether the C-string is null
         JE    NullData           Null string, no nonblanks
         BCTR  5,0                Back up GR5 to the byte before null
         LA    2,CData            Set GR2 to starting address
         LR    1,5                Copy end address to GR1
         SR    1,2                Find length to search for nonblank
         EX    1,TRTR             Use TRTR to search for a nonblank
         JZ    AllBlank           No nonblank character was found
         - - -                    GR1 points to the nonblank
TRTR     TRTR  0(*-*,5),NonBlank  Find a nonblank character
NonBlank DC    256X'1'            Fill the table with nonzero bytes
         ORG   NonBlank+C' '      Position at offset X'40'
         DC    X'0'               Don't stop on blanks
         ORG   ,                  Adjust Location Counter
```

25.4.2. Consider these instructions:

```
         XR    0,0               Search argument is a null character
         LA    7,WorkArea+L'WorkArea-1      End of search area
         LA    4,WorkArea        Start of search area
Search   SRST  7,4               Search for the terminating null
         JO    Search            Repeat if not found yet
         JH    NotFound          No terminating null found?
         LA    0,WorkArea        Starting address
         SR    7,0               Null byte address - start address
         ST    7,WorkLen         Store the C-string's length
```

The JH instruction illustrates a way to check for the possibility of invalid C-string data at **WorkArea**. Taking immediate remedial action is better than finding the address of a random null character elsewhere in memory and assigning that as the length of the C-string. (The next operation might try to move the "C-string" to a new work area, possibly over-writing important parts of your program.)

25.4.3. Be careful! You can't immediately use the SRST instruction, because it won't necessarily stop after N characters have been tested. You should first set the address in R_1 to the address of (A+c(m)), where m > min(c(N),strlen(WorkArea)). Use your solution to Exercise 25.4.2 to find strlen(WorkArea).

25.4.4. Two searches will be needed:

1. a first search to find a null character to determine the string's length,
2. a second to find the character at OddByte

What will happen if the string at Clutter is a null string?

Section 25.5

25.5.1. If you had written

```
         MVC   New,Old
```

you would first have to know that the length of the characters at **Old** was fewer than 255 bytes long. If this is true, you would have correctly moved the string *and* the null terminator, because the Length Specification byte would be derived from the length attribute of **New**. However, if you had written

```
         MVC   New(L'Old),Old
```

the terminating byte would not have been moved.

25.5.3. It contains the token's length without the comma. Consider the first token, LIST. The terminating comma is at **Source + 4** so that when the starting address of **Source** is subtracted, the token length is 4.

25.5.4. This solution first finds the length of the From string to test whether it will fit within N bytes.

```
         XR    0,0               Search for terminating null byte
         LA    1,From            Point to start of search string
         LR    2,1               Copy address
         AL    2,NBytes          Address of first byte past valid area
RepeatS  SRST  2,1               Test for null terminator byte
         JO    RepeatS           Branch if not found yet
         JL    Okay              Branch if From string will fit
         XR    1,1               From string is too long to fit
         L     3,NBytes          Length of From to be moved
         LR    5,3               Length of Target to be moved into
         LA    2,Target          Address of target string
         LA    4,From            Address of source string
         MVCL  2,4               Move N bytes
         J     Done              Finished with the move
Okay     LA    1,Target          Address of Target string
         LA    2,From            Address of source string
RepeatM  MVST  2,1               Move some bytes
         JO    RepeatM           Move some more bytes if necessary
         AHI   1,1               Point GR1 past the final byte
Done     - - -
```

25.5.5.

```
         XR    0,0                 Search for terminating null byte
         LA    1,Prefix            Address of Prefix string
         LR    2,1                 Copy address
         AHI   2,8000              Maximum length of Prefix string
Srch     SRST  2,1                 Search for terminating null
         JO    Srch                Repeat if not found yet
         JH    Error               Null not found within 8000 bytes!
*        GR1 = address of the null byte terminating Prefix string
         LA    3,Suffix            Address of Suffix string
Move     MVST  1,3                 Move Suffix string after Prefix
         JO    Move                Repeat if necessary
         - - -
```

25.5.6. If you completed Exercises 25.5.4 and 25.5.5, the solution to this exercise is straightforward.

Section 25.6

25.6.1. Consider these instructions:

```
         XR    0,0                 Both strings terminated by null bytes
         LA    1,StringA           Address of StringA
         LA    2,StringB           Address of StringB
Test     CLST  1,2                 Compare the two strings
         JO    Test                Equal so far, repeat comparison
         JE    Done                Strings are equal, c(GR0)=0
         JL    ALow                Jump if StringA < StringB
         AHI   0,1                 Indicate StringA > StringB, c(GR0)=+1
         J     Done                Finished
ALow     BCTR  0,0                 StringA < StringB, c(GR0)=-1
Done     - - -
```

25.6.3.

```
(1)   CC1,  c(GR1)=X'26F945',  c(GR2)=X'26F94A'
(2)   CC0,  c(GR1)=X'26F945',  c(GR2)=X'26F946'
(3)   CC1,  c(GR1)=X'26F947',  c(GR2)=X'26F94C'
(4)   CC2,  c(GR1)=X'26F945',  c(GR2)=X'26F94D'
```

Section 25.7

25.7.1. Undoubtedly something unpleasant.

25.7.2. The results are shown in the following table:

c(GR2)	c(GR3)	CC
X'7F290F'	4	1
X'7F2913'	0	0
X'7F2914'	42	1

Section 25.8

25.8.2. 255, because the substring length is in the rightmost byte of GR0.

25.8.3. If CUSE restarts at the second equal byte, it will discover the same inequality one byte earlier. Thus it should restart with the bytes following the unequal bytes.

25.8.4. No. Both strings have the same length (4 bytes), and the search for matching substrings stops at the end of the longer operand.

25.8.5. CLCLE searches for the first pair of unequal bytes (padding the shorter operand if necessary), while CUSE searches for the first pair of equal bytes matching the padding byte.

Section 26 Solutions

Section 26.1

26.1.1. Because BCD characters can take on only 64 values, your translation or mapping table needs only 64 entries. Here's a typical table, where octal values not representing characters are translated to blanks:

```
          DC    C'0123456789 =''   '    Values 00-17 octal
          DC    C'+ABCDEFGHI .)    '    Values 20-37 octal
          DC    C'-JKLMNOPQR $*    '    Values 40-57 octal
          DC    C' /STUVWXYZ ,(    '    Values 60-77 octal
```

26.1.3. The octal digits are

```
60 60 60 60 60 60 60 60 60 51 25 63 64 51 45 60 60 60 60 60
```

26.1.5. 2^{N-1}.

Section 26.2

26.2.1. Some other variant characters are:

<table>
<tr><th>Page</th><th>X'4A'</th><th>X'5A'</th><th>X'9F'</th><th>X'A1'</th><th>X'B0'</th><th>X'B1'</th><th>X'BA'</th><th>X'BB'</th></tr>
<tr><td>037</td><td>¢</td><td>!</td><td>¤</td><td>~</td><td>^</td><td>£</td><td>[</td><td>]</td></tr>
<tr><td>1143</td><td>§</td><td>∈</td><td>]</td><td>ü</td><td>£</td><td>[</td><td>¬</td><td>|</td></tr>
</table>

The ¤ symbol was a technical and political compromise meant to represent the local currency symbol in each country. Also, the ∈ symbol is meant to represent the European “Euro”, but these notes only have the mathematical “member of” symbol available as the closest approximation.

26.2.2. If your source program is encoded using code page 037 (the Assembler's default), the generated machine language is X'7C7B5B4FC0BABBD0'. However, if the code page used for your source program is different (possibly one of those in Table 171 on page 431), you might find that the Assembler generates different values.

26.2.3. 82.

Section 26.3

26.3.1. Alphanumeric characters in the ASCII character set have representations less than X'80', but in EBCDIC they have representations greater than X'80'.

26.3.2. The generated data is:

```
Con1:  X'352A28322E3233362F44656E6F6D292B5069'
Con2:  X'496E76616C69642045787072657373696F6E3F'
Con3:  X'48656C6C6F2C20576F726C6421'
Con4:  X'4269744D61736B26427974657C7C27436861727327'
```

Section 26.4

26.4.1. Because the DBCS option is active, the third statement causes an error because the explicit length truncates into the middle of DBCS data.

```
A         DC    G'<.A.B.C>'        generates X'42C142C242C3'
B         DC    GL4'<.A.B.C>'                X'42C142C2'
C         DC    CL4'<.A.B.C>'      error, truncates into DBCS data
D         DC    C'<.A.B.C>'        generates X'0E42C142C242C30F'
```

Section 26.5

26.5.1. There would be no changes, because all of the characters are among the ASCII characters in the first 127 bytes of the Unicode Basic Multilingual Plane.

26.5.3. The generated data is

```
C1        X'004100420043'
C2        X'004100420043002000200020'
C3        X'0041'
C4        X'0041002600420027'
```

Section 26.6

26.6.2. One operand can be padded, but not both; only the shorter operand will be padded.

26.6.5. You can use only those Unicode characters with encodings between U+0000 and U+0FFF, because the displacement of those two instructions is limited to 12 bits.

26.6.6. The instruction LA MapTbl-X'4040' could fail because the required displacement might be negative. Some alternatives are

```
LAY   1,MapTbl-X'4040'
LARL  1,MapTbl-X'4040'
L     1,=A(MapTbl-X'4040')
```

but the literal in the last instruction must be addressable!

26.6.7. Yes, because no test character is involved when the optional operand is 1.

26.6.8. It depends. If the optional operand is 0, no, because the test character is two bytes long rather than one byte. If the length is even and the optional operand is 1, yes.

26.6.9. Consider the bit patterns in the first byte:

Bit Pattern	Total Bytes
0xxx xxxx	1
10xx xxxx	Not a starting byte!
110x xxxx	2
1110 xxxx	3
1111 0xxx	4

If a byte starts with B'10', you can scan backward or forward at most three bytes to find one of the valid starting patterns. This is sometimes called the “self-synchronizing” property of UTF-8.

26.6.12. Here is the required translate table. It was found on the IBM “Globalization” web site.

```
*        SBCS-to-Unicode mapping for 037 to Unicode 10646
         DC    X'0000,0001,0002,0003,009C,0009,0086,007F'
         DC    X'0097,008D,008E,000B,000C,000D,000E,000F'
         DC    X'0010,0011,0012,0013,009D,0085,0008,0087'
         DC    X'0018,0019,0092,008F,001C,001D,001E,001F'
         DC    X'0080,0081,0082,0083,0084,000A,0017,001B'
         DC    X'0088,0089,008A,008B,008C,0005,0006,0007'
         DC    X'0090,0091,0016,0093,0094,0095,0096,0004'
         DC    X'0098,0099,009A,009B,0014,0015,009E,001A'
         DC    X'0020,00A0,00E2,00E4,00E0,00E1,00E3,00E5'
         DC    X'00E7,00F1,00A2,002E,003C,0028,002B,007C'
         DC    X'0026,00E9,00EA,00EB,00E8,00ED,00EE,00EF'
         DC    X'00EC,00DF,0021,0024,002A,0029,003B,00AC'
         DC    X'002D,002F,00C2,00C4,00C0,00C1,00C3,00C5'
         DC    X'00C7,00D1,00A6,002C,0025,005F,003E,003F'
         DC    X'00F8,00C9,00CA,00CB,00C8,00CD,00CE,00CF'
         DC    X'00CC,0060,003A,0023,0040,0027,003D,0022'
         DC    X'00D8,0061,0062,0063,0064,0065,0066,0067'
         DC    X'0068,0069,00AB,00BB,00F0,00FD,00FE,00B1'
         DC    X'00B0,006A,006B,006C,006D,006E,006F,0070'
         DC    X'0071,0072,00AA,00BA,00E6,00B8,00C6,00A4'
         DC    X'00B5,007E,0073,0074,0075,0076,0077,0078'
         DC    X'0079,007A,00A1,00BF,00D0,00DD,00DE,00AE'
         DC    X'005E,00A3,00A5,00B7,00A9,00A7,00B6,00BC'
         DC    X'00BD,00BE,005B,005D,00AF,00A8,00B4,00D7'
         DC    X'007B,0041,0042,0043,0044,0045,0046,0047'
         DC    X'0048,0049,00AD,00F4,00F6,00F2,00F3,00F5'
         DC    X'007D,004A,004B,004C,004D,004E,004F,0050'
         DC    X'0051,0052,00B9,00FB,00FC,00F9,00FA,00FF'
         DC    X'005C,00F7,0053,0054,0055,0056,0057,0058'
         DC    X'0059,005A,00B2,00D4,00D6,00D2,00D3,00D5'
```

```
         DC    X'0030,0031,0032,0033,0034,0035,0036,0037'
         DC    X'0038,0039,00B3,00DB,00DC,00D9,00DA,009F'
```

You would probably use the TROT instruction.

Note that all the characters in code page 037 translate to UTF-16 characters starting with `X'00'`, so they are included in the Basic Mutlilingual Plane (BMP).

26.6.13. This elegant solution and its analysis are due to John Ganci.

```
***********************************************************************
* Generate a 512-byte table of all 2-byte printable hexadecimal      *
* values corresponding to every possible 1-byte binary value.        *
* For example, the table entry for X'00' is X'F0F0' and the table    *
* entry for X'9C' is X'F9C3'.                                        *
*--------------------------------------------------------------------*
* Numeric digits in the following discussion are all base ten.       *
* Let h be a hexadecimal digit (0 <= h <= 15). Let t be the          *
* 10's digit of h when h is written in base 10 and let u be          *
* the unit's digit of h when h is written in base 10. Then           *
*                                                                    *
*   h = 10 * t + u                                                   *
*                                                                    *
* Note that t is 0 or 1 and                                          *
*                                                                    *
*   0 <= u <= 9 when t = 0,                                          *
*   0 <= u <= 5 when t = 1.                                          *
*                                                                    *
* The printable hexadecimal byte for h is                            *
*                                                                    *
*   240 + u = 240 + h = X'F0' + u   if  0 <= h <= 9                  *
*   193 + u = 183 + h = X'C1' + u   if 10 <= h <= 15                 *
*                                                                    *
* In either case, the printable hexadecimal byte for h is            *
*                                                                    *
*   193 + 47(1 - t) + u = 240 - 47t + u                              *
*--------------------------------------------------------------------*
* Let v = X'hk' be a source byte value represented in hexadecimal.   *
*                                                                    *
* We compute PH(v) = 2-byte Printable Hexadecimal value of v.        *
*                                                                    *
* The value is computed as a 2-byte entry in a table, with the       *
* table entry for v at offset 2v in the table.                       *
*                                                                    *
* (1) v = (*-ph)/2 = h * 16  +  k                                    *
*                                                                    *
* (2) 16's digit = h = (1) / 16 = ((*-ph)/2)/16                      *
*                                                                    *
* (3) 1's digit = k = (1) - (2) * 16                                 *
*                   = (*-ph)/2-(((*-ph)/2)/16)*16                    *
*                                                                    *
* (4) 10's digit for h = r = h/10 = (((*-ph)/2)/16)/10               *
*                                                                    *
* (5) 1's digit for h = s = h - (h/10)*10                            *
*                         = ((*-ph)/2)/16-((((*-ph)/2)/16)/10)*10    *
*                                                                    *
* (6) 10's digit for k = t = k/10 = ((*-ph)/2-(((*-ph)/2)/16)*16)/10  *
*                                                                    *
* (7) 1's digit for k = u = k - (k/10)*10                            *
* = (*-ph)/2-(((*-ph)/2)/16)*16-(((*-ph)/2-(((*-ph)/2)/16)*16)/10)*10 *
*                                                                    *
* (8) PH(v) = printable hex value = (240-47r+s)*16*16+240-47t+u      *
***********************************************************************
         Print Data               For validating the generated data
         SPACE 1
PH       DS    0D
         DC    256AL2((240-47*((((*-PH)/2)/16)/10)+((*-PH)/2)/16-((((*-X
               PH)/2)/16)/10)*10)*16*16+240-47*(((*-PH)/2-(((*-PH)/2)/1X
```

```
                6)*16)/10)+(*-PH)/2-(((*-PH)/2)/16)*16-(((*-PH)/2-(((*-PX
                H)/2)/16)*16)/10)*10)
```

26.6.14. It's probably easiest to use UTF-8 as an intermediate representation.

For all but surrogate pairs, UTF-16 characters can be transformed to UTF-32 by adding two zero high-order bytes. For mapping UTF-16 surrogate pairs to UTF-32, the bit mappings in Figures 252 and Figure 253 on page 447 show how to convert the bits.

To transform UTF-32 characters to UTF-16, if the two high-order bytes are zero simply remove them to create the UTF-16 format. If there are nonzero bits in the two high-order bytes, again consult the figures in Section 26.5.

26.6.15. See the solution to Exercise 26.6.9 above.

Section 26.7

26.7.1. The function-code table can be either 256 or 512 bytes long. Because you are translating only character values less than 255, the choice of table size depends on the range of values you want to assign to the function codes. If all function codes are less than 255 you can use either table size; but if some function codes lie between 256 and 65535, you must use a 512-byte function code table.

26.7.2. One elegant solution[333] is

```
TL       Equ   (F+1)*(256+A*(1-L)*5280)
```

An alternative, but much more complex solution is[334] is

```
TL       Equ   (((((A+1)/A)/2)*256+((1-A)/(1+A)))*(F+1)*256)*((1-(A*L))=
               /(1+(A*L)))+(A*L)*(F+1)*256
```

Some terms may be worth explaining. In the last addition,

```
               (A*L)*(F+1)*256
```

will be evaluated as 256 or 512 only if both A and L are 1. This takes care of the last two rows of Table 189 on page 450. Similarly,

```
               ((1-(A*L))/(1+(A*L))
```

is 1 only if either A or L is zero. This takes care of the first four rows of the table. The remaining subexpressions calculate the other four values of TL.

26.7.5. Among the similarities and differences:

1. TRTE translates and tests without changing the first operand; the other four only translate.
   - TROO is similar to TRTE with A=0, F=0
   - TROT is similar to TRTE with A=0, F=1
   - TRTO is similar to TRTE with A=1, F=0, L=0
   - TRTO is similar to TRTE with A=0, F=0, L=1, with no nonzero bits in the high-order argument byte
   - TRTT is similar to TRTE with A=1, F=1, L=0
   - TRTT is similar to TRTE with A=1, F=1, L=1. with no nonzero bits in the high-order argument byte

26.7.6. The needed modifications to the example in Figure 255 on page 451 are straightforward.

```
Z        Equ   F+1                 Argument character length
*
Tbl      DC    0D,(X'28')AL(Z)(0)
         DC    AL(Z)(LP-B)         Left parenthesis
         DC    AL(Z)(RP-B)         Right parenthesis
         DC    AL(Z)(M-B)          * (Multiplication)
         DC    AL(Z)(P-B)          + (Addition)
         DC    AL(Z)(0)            Ignored character
         DC    AL(Z)(S-B)          - (Subtraction)
         DC    AL(Z)(0)            Ignored character
         DC    AL(Z)(D-B)          / (Division)
         DC    10AL(Z)(N-B)        Numeric digit
         DC    (X'FF'-X'39')AL(Z)(0)   Ignored characters
```

333 Due to Robert Netzlof.

334 My own.

Section 26.8

26.8.1. Suppose the bytes in GR2 are labeled wxYZ (from left to right). The LRVR instruction changes the byte order to ZYxw; the SRA instruction then shifts ZY right, propagating the leftmost bit of Z as the sign of the 32-bit result. Replacing SRA by SRL leaves an unsigned value in GR2.

26.8.3.

```
         LRV   1,DPG
```

Compare this one instruction to those in your solution to Exercise 17.2.5.

Section 27 Solutions

Section 27.1

27.1.1. Both instructions in Figure 263 on page 461 use implied lengths, relying on the length attribute of the target operand. Thus, they move 8 zones or digits. If the fields are defined as 8X'00' their length attribute is 1, and only one zone or numeric will be moved. Thus, the results would be

```
c(Numerics) = X'0E'
c(Zones)    = X'F0'
```

27.1.2. The result will be the same as executing

```
MVC   Target,Source
```

except that it will take more time.

27.1.3. The key to this solution is the translate table, which is arranged like this:

```
  0             9 A       F
  ┌─────────────┬───────┐
 0│             │       │
  │  Bad Sign   │OK Signs│
  :             :       :
 9│             │       │
  ├─────────────┼───────┤
 A│  Bad Byte   │ Bad   │
  │             │ Digit │
  :             :       :
 F│             │       │
  └─────────────┴───────┘
```

```
        XR    2,2
        TRT   ZTest+L'ZTest-1(1),ZDLast
        B     *(2)
        J     BadSign
        J     ZPlus
        J     ZMinus
        J     BadDigit
        J     BadByte
        - - -
*       4=Bad Sign, 8=Plus, 12=Minus, 16=Bad Digit, 20=Bad Byte
ZDLast  DC    10AL1(4,4,4,4,4,4,4,4,4,4,8,12,8,12,8,8)
        DC    6AL1(20,20,20,20,20,20,20,20,20,20,16,16,16,16,16,16)
```

You might try extending the translate table to detect a value of ±0 in the last byte, and branch to **Plus0** and **Minus0** as appropriate. (The TP instruction described in Section 29.1 is much simpler!)

27.1.4. The translate table checks for zoned numerics:

```
        TRT   ZTest(L'ZTest-1),ZDNums
        JNZ   BadZDig             Condition Code not zero
        - - -
ZDNums  DC    240AL1(4)           X'00'-X'EF' invalid
        DC    10AL1(0)            X'F0'-X'F9' valid
        DC    6AL1(4)             X'FA'-X'FF' invalid
```

27.1.5. The printed characters will be E, N, C, and R.

27.1.6. For EBCDIC, the rows punched for the + character are 12-8-6, and for the − character only row 11. For BCD, the − character is also row 11 only, and the + character is row 12 (the top row).

27.1.7. Since only the zone portion of the byte is being moved, any five-byte literal with X'F' zone digits would work.

27.1.8. The second form is incorrect because the duplication factor is not an absolute expression.

Section 27.2

27.2.1. *All* of the constants are 5 bytes long, so the generated data for the last two constants will be different.

27.2.2. The values are (1) `X'D1'`, (2) `X'C0'`, (3) `X'F0F4D2'`, (4) `X'F0C5'`.

27.2.3. The values are (1) −09, (2) +2, (3) +007, (4) −0.

27.2.4. The generated values are (1) `X'F0F0F0F0F0F0F0F1F2F3F4F5F6F7F8C9'` (7 leading zeros), (2) Length error (too many digits), (3) `X'F5F4F3F2C1'` (high-order digit truncated), (4) Error (length modifier greater than 16).

27.2.5. In the rightmost byte, 6×10=60; in any other byte, 10. In the rightmost byte, the 10 decimal digits can be paired with any of ABCDEF for the sign digit; remember that negative zero is valid.

27.2.6. Because the constant is type C, only the first 5 characters will be present in the assembled constant! (Remember, the commas are part of the nominal data.)

27.2.7. Both have Length Attribute 5.

27.2.8. I+S=L=N.

Section 27.3

27.3.1. The integer part of (N+2)/2, or (N/2)+1.

27.3.2. In the rightmost byte, 6×10=60; in any other byte, 10×10=100.

27.3.3. The possible bit combinations in each byte of the packed decimal number at **Pack** need to be analyzed to determine where they might occur. This diagram divides the possible byte values into four groups:

	0-9	A-F
0 9	Box A	Box B
A F	Box C	Box D

Box A contains bytes all of whose digits are 9 or less; Box B contains bytes whose first digit is 9 or less and whose second digit is `X'A'` or greater. Box C contains bytes whose first digit is greater than `X'A'` and whose second digit is 9 or less. The remaining bit combinations are in Box D.

By examining the contents of each box, we can use the Condition Code setting after TRT to distinguish many cases:

Box	CC=0	CC=1	CC=2
A	All bytes contain only numeric digits so the number has an invalid sign.	A byte before the last contains a second digit greater than 9, so it has an invalid digit.	The last byte has a valid numeric digit but an invalid sign.
B	Should not occur.	Sign digit before the last byte is invalid.	Valid positive or negative data.
C	Should not occur.	An invalid digit appears in a byte before the last.	The last byte contains an invalid digit and sign.
D	Should not occur.	A byte before the last contains an invalid digit.	The last byte contains an invalid digit.

Now, we can construct and use the required translate table:

```
         LA    1,Pack+L'Pack-1    Point to last byte of Pack (for CC=0)
         XR    2,2                Set GR2 to zero
         TRT   Pack,PDTest        Test the packed data
         B     J0(2)              Branch to check resulting CC
*
J0       J     BadSign            All numerics, sign is invalid
*
P        Equ   *-J0               Box B Offset for + result
         JZ    Error              CC=0: something went wrong
         JM    BadDigit           CC=1: sign before last digit
         J     PPlus              CC=2: + sign on valid number
*
M        Equ   *-J0               Box B Offset for - result
```

```
         JZ    Error             CC=0: something went wrong
         JM    BadDigit          CC=1: sign before last digit
         J     PMinus            CC=2: - sign on valid number
*
C        Equ   *-JO              Box C offset
         JZ    Error             CC=0: something went wrong
         JM    BadDigit          CC=1: bad digit
         JP    BadByte           CC=2: bad sign and digit
*
D        Equ   *-JO              Box D offset
         JZ    Error             CC=0: something went wrong
         J     BadDigit          CC=1,2: bad digit
*
PDTest   DC    0X
Row0_9   DC    10AL1(0,0,0,0,0,0,0,0,0,0,P,M,P,M,P,P) 1st digit 0-9
RowA_F   DC    6AL1(C,C,C,C,C,C,C,C,C,C,D,D,D,D,D,D) 1st digit A-F
```

27.3.4. One way to do this is with

```
         NI    SomeVal+L'SomeVal-1,X'FE'    Force last bit to 0
```

Any sign code with rightmost bit zero (A, C, E) indicates a plus sign.

27.3.6. It can't be done with a single instruction. (Compare this exercise to Exercise 27.3.4.) Sign codes X'B' and X'D' indicate a negative value, but X'F' indicates a positive value. With two instructions, it's easy. (Show how!)

Section 27.4

27.4.1.

- c(Z1) = X'F2F1F4F7F4F8F3F6F4F8'
- c(Z2) = X'F1F4F7F4F8F3F6F4F7' (truncation of one digit)
- c(P3) = X'99999999999D'
- c(P4) = X'123C456C789C' (three separate constants);
- The explicit length of 20 is illegal for both the packed and the zoned constants.
- c(P6) = X'31415926535C' (decimal point ignored)
- c(P7) = X'0CF0' (both constants truncated)
- c(Z8) = X'F1F2F0F4F0F0F8F0...F0'

27.4.2.

- L'Z1 = 10
- L'Z2 = 9
- L'P3 = 6
- L'P4 = 2 (!)
- L'P6 = 6
- L'P7 = 1
- L'Z8 = 1

27.4.3. The generated constant, X'000D', is truncated.

27.4.4. No. The Assembler complains only if the number of digits in the nominal value is greater than 31, even if removing leading zero digits would leave fewer than 32 significant digits! (Do you think the Assembler should be less – or more – fussy?)

27.4.5. Because signed packed decimal numbers always have an odd number of digits, the high-order zero digits in the nominal values are the same as the (default) zero digits inserted by the Assembler to ensure an odd number of digits.

27.4.6. This can be done with an EX instruction:

```
         XR    6,6               Set GR6 to zero
         LA    1,L'P             Length of the packed operand
         IC    1,T-1(1)          Insert appropriate mask bits
         EX    1,ICM             Insert bytes at right end of GR6
         - - -
ICM      ICM   6,*-*,P           Executed to insert 1-4 bytes
T        DC    X'1,3,7,F'        Masks for ICM instruction
```

27.4.8. Four possible interpretations, and their Assembler Language defining statements, are:

```
         DC    F'1077952604'       Fullword binary integer
         DC    C'•••*'             Characters (three blanks and an asterisk)
         STH   4,X'05C'(0,4)       Store Halfword
         DC    P'+4040405'         Packed decimal
```

Further interpretations are possible, as we'll see later.

27.4.9. This solution is due to John Ganci:

```
         Print Data                To verify all data is generated
S1       DS    0D
         DC   1000AL2((((*-S1)/2)/100)*16*16*16+(((*-S1)/2-(((*-S1)/2)/X
               100)*100)/10)*16*16+(((*-S1)/2-(((*-S1)/2)/100)*100)-(((X
               (*-S1)/2-(((*-S1)/2)/100)*100)/10)*10))*16+12)
```

To describe this solution, he writes:

1. Let V = (*-S1)/2 = H*100 + T*10 + U (Hundreds, Tens, and Units)
2. H = V/100 = ((*-S1)/2)/100
3. Remove the Hundreds digit: TU = T*10+U = V-H*100 = (*-S1)/2-(((*-S1)/2)/100)*100
4. T = TU/10 = ((*-S1)/2-(((*-S1)/2/100)/100)*100)/10
5. Remove the Tens digit: U = TU - T*10 = U - (TU/10)*10 = (*-S1)/2-(((*-S1)/2)/100)*100-(((*-S1)/2-(((*-S1)/2/100)/100)*100)/10)*10
6. Then, generate the packed decimal constant `X'HTUC' =` H*16*16*16+T*16*16+U*16+12

What will happen if you replace the constant type `AL2` with `Y`? Try it!

27.4.10. Like the solution to Exercise 27.4.9, this is also due to John Ganci:

```
         Print Data                To verify all data is generated
Z3       DS    0D
         DC   1000AL3((((*-Z3)/3)/100)*16*16*16*16+(((*-Z3)/3-(((*-Z3)/X
               3)/100)*100)/10)*16*16+(((*-Z3)/3-(((*-Z3)/3)/100)*100)-X
               ((((*-Z3)/3-(((*-Z3)/3)/100)*100)/10)*10))+X'F0F0C0')
```

The analysis of this exercise is similar to that in Exercise 27.4.9, but the generated data doesn't line up as neatly in the listing.

27.4.11. The Scale Attribute values are: `S'A=7, S'B=3, S'C=1, S'D=4, S'E=0.`

27.4.12. The number of packed decimal digits is `2*L'X−1`, so the value of the Integer Attribute is `I'X = 2*L'X−1−S'X`. That is, the total number of digits minus the number of fraction digits.

27.4.13. The attributes of the symbols are:

Symbol	L'	I'	S'
A	5	2	7
B	5	6	3
C	5	8	1
D	4	3	4
E	6	11	0

27.4.15. L=(N/2)+1; N=(2L-1)=I+S.

Section 27.5

27.5.1.

```
         PACK  AA(5),BB(5)         F244 ---- ----
         UNPK  BB,AA               F3A1 ---- ----
         PACK  0(16,9),65(,2)      F2F0 9000 2041
         PACK  AA(0),BB(0)         F200 ---- ----
         UNPK  BB-AA(,9),BB(L'AA)  F3A1 9002 ----
         PACK  AA,BB+11(3)         F212 ---- ----
```

Section 27.6

27.6.1. NP = (NZ+2)/2, or (NZ/2)+1.

27.6.2. The results for the three operands are:

1. X'0000000FFF7F'
2. X'336B066934A5'
3. X'CCFC07F2C412'

27.6.4. The address of the rightmost byte of the first (target) operand must be greater than or equal to the address of the rightmost byte of the second (source) operand. This is perhaps deceptive: it might seem that the first operand's rightmost byte address could be as much as 2 less than the second operand's rightmost byte address, since “the result byte is stored immediately after fetching the necessary operand bytes”. However, the fetched bytes are those for the *current* result byte, not the *next* result byte!

27.6.5.

- c(P1) = X'468C'
- c(P2) = X'02468C'
- c(P3) = X'6C'
- c(P4) = X'00000681357C'
- c(P5) = X'002468135F'

27.6.6. The result is correct for any length, so the maximum length is 16 and the minimum length is 1.

27.6.7. X'000000013579BDFE'.

27.6.8. Packing a string of blanks gives X'000...004', which has an invalid sign code for arithmetic purposes. The rightmost byte could be corrected with an OI instruction with mask X'0C'. (Note also that ORing with a mask X;08' would work when packing any field containing *either* numeric digits or all blanks!)

27.6.9. The result will be valid for any value of K satisfying $1 \leq K \leq N$.

27.6.10. The length digit for **PackData** is in the L_1 field, so it must be positioned correctly:

```
         BCTR  2,0                 Reduce count for EXecute
         SLL   2,4                 Position the count digit correctly
         EX    2,PackOp            Do the Pack instruction
         - - -
PackOp   Pack  PackData(*-*),ZData     Executed instruction
```

27.6.11. The result will be X'00BDFE'.

27.6.12. The result will be X'00BEFE'.

27.6.13. The result will be X'00BDFE'.

27.6.14. There is no difference so long as there are one or more zoned decimal digits at the right end of the source operand, because both zeros and blanks have numeric digit zero.

27.6.15. The results are

```
(1)  Source = C'34567', Packed = X'0034567F' (valid)
(2)  Source = C'ABCDE', Packed = X'0012345C' (valid)
(3)  Source = C'****0', Packed = X'00CCCC0F' (invalid digits)
(4)  Source = C'$2.98', Packed = X'00B2B98F' (invalid digits)
(5)  Source = C'     ', Packed = X'00000004' (invalid sign)
(6)  Source = C'VWXYZ', Packed = X'0056789E' (valid)
```

Examples (2) and (6) illustrate the dangers of “successfully” PACKing fields containing character data.

Section 27.7

27.7.1. c(PackOp) = X'12345C'; c(ZonOp) = X'F1F2F3F4C5'.

27.7.2. The results are:

1. X'F0F0FFFFFFFFFFFFF7F'
2. X'F9F6F9F9F9F3F8F4A5'
3. X'F6FFF7F2F6FCF6F412'

27.7.3. X'FFFFFCFFFCFCFDFE'. (Were you surprised?)

27.7.5. If the length of either operand is 1 byte, there is no problem with overlap.

Now, assume neither operand is one byte long. Then, the address of the rightmost byte of the first (zoned) operand must be greater or equal to than the address of the rightmost byte of the second (packed) operand plus the length of the second operand, minus 2. That is, if Z_L is the address of the last byte of the zoned operand, and P_L for the second operand, then

$$Z_L \geq P_L + L'P - 2$$

where L'P is the length of the packed second operand. This avoids the possibility that the byte pairs of the first operand being stored before single bytes are fetched for the second operand.

27.7.6.

- c(Z1) = X'F1F2F3F4F5C6'
- c(Z2) = X'F0F1F2F3F4F5C6'
- c(Z3) = X'F5C6'
- c(Z4) = X'F0F0F0F4F5C6'
- c(Z5) = X'F1F2F354'.
- The contents of Z6 is unknown, since the implied length (4) of **PData** means that unpacking will start with the byte at PDATA+6. If we assume that the byte at PData+3 contains the two hex digits X'xy', then c(Z6)=X'FCyx'.

27.7.7. The minimum length is of course 1; the maximum length is 3. (See the solution to Exercise 27.7.3 for an illustration.)

27.7.8. NZ = 2×NP−1. They are different because we discarded the remainder in evaluating NP=(NZ/2)+1.

27.7.11. The result is X'FFFFFFFFFFFFFDEF'

27.7.12. Use implied lengths for the second operand of PACK and UNPK.

27.7.14. The result will be X'F0F1F2F3F465'.

27.7.15. The result will be X'F0FFF2F3F465'.

27.7.16. The result will be X'F0FFF5F6F565'.

27.7.17. The result at **Answer** will be X'F0F0F7F6F5F4C3'.

Section 27.8

27.8.1. The result is 16 bytes long: X'00000000 12345678 90123456 789ABCDC' (the spaces are inserted to improve readability).

27.8.2. The result will be X'0BDFBDFBDFBDFBDFBDFBDFBDFBDFBDFC'.

27.8.3. The result will be X'0DBFDBFDBFDBFDBFDBFDBFDBFDBFDBFC'.

27.8.4. Because the required 16 bytes of the packed decimal operand would overlap the source operand, the results are unpredictable. (Try it, and see what happens!)

27.8.5. Assuming that no test is made for all unpacked characters being zeros, CC1 should indicate a negative packed operand, and CC2 should indicate a positive packed operand. If the test for zero characters was made, CC0 should be set. (Question: should such a test for zeros check the entire packed operand, or only the digits actually unpacked?)

27.8.6. X'0000 0000 0000 0000 0000 0009 8765 432C', where the spaces were inserted for readability.

27.8.7. The CC setting will be zero in both cases!

Section 27.9

27.9.1. Because all of the bytes being translated have values between X'F0' and X'FF', we must ensure that the first byte of the translate table at **TRTab** is accessed if the source byte is X'F0'.

27.9.2. X'0F', from swapping the digits of the first byte of the translate table.

27.9.3. Be careful! It is tempting to start by writing

```
UNPK  String(17),DW(9)
```

only to discover that the length of 17 is invalid. The unpacking must be done in two steps, as in

```
         UNPK  String(9),DW(5)       First 4 bytes
         UNPK  String+8(9),DW+4(5)   Last 4 bytes
         TR    String(16),=C'0123456789ABCDEF'-X'F0'  Translate
```

Does it matter in which order the above two UNPK's are executed? Try it, and then explain why or why not.*

27.9.4. The second operand was exhausted before the first operand field was filled.

27.9.5. Suppose a (non-zero) register (say, 7) is free. Then rewrite the TR either as

```
         L     7,=A(TRTab-X'F0')  Point to offset from table
         TR    Char(8),0(7)       Use the same translate table
```

Of course, the literal must be addressable! If not, use LY instead of L. Or, write

```
         LAY   7,TRTab-X'F0'      Point to offset from table
         TR    Char(8),0(7)       Use the same translate table
```

If neither of these methods work, consider rewriting that part of your program.

27.9.6. This elegant technique is due to D. R. Page.** The code is surprisingly simple:

```
         UNPK   BinaryCh(3),Byte(2)
         TR     BinaryCh(2),TBL-X'F0'
         UNPK   BinaryCh(5),BinaryCh(3)
         TR     BinaryCh(4),TBL-X'F0'
         UNPK   BinaryCh(9),BinaryCh(5)
         - - -
Byte     DC     B'10101101'       Input Data
         DS     C                 Work byte
BinaryCh DS     CL8,C             Result string + work byte
TBL      DC     X'00010405101114154041444550515455'
```

27.9.7. The solution is obtained by “inverting” the solution to Exercise 27.9.6. The translate table is different, however: although only 16 bytes are used for the translation, an area 86 bytes long is required. (The gaps could be filled with other data if you like; “Q” can represent any byte value, because those bytes of the table are not referenced.)

```
Q        Equ   0                  Or any other value < 256
         PACK  BinaryCh+4(5),BinaryCh(9)
         TR    BinaryCh+4(4),Table
         PACK  BinaryCh+6(3),BinaryCh+4(5)
         TR    BinaryCh+6(2),Table
         PACK  BinaryCh+7(2),BinaryCh+6(3)
         MVC   Byte(1),BinaryCh+7 Move result byte
         - - -
BinaryCh DC    CL8'10101101',C' ' Input data + work byte
Byte     DS    X                   Result byte
Table    DC    AL1(0,1,Q,Q,2,3),10AL1(Q)
         DC    AL1(4,5,Q,Q,6,7),10AL1(Q)
         DC    32AL1(Q)
         DC    AL1(8,9,Q,Q,10,11),10AL1(Q)
         DC    AL1(12,13,Q,Q,14,15)    (86 bytes in all!)
```

* The order *does* matter. If the UNPKs are reversed, the byte at String+8 will contain the swapped digits of the byte at DW+5.

** IBM Technical Disclosure Bulletin, Volume 18, January 1973, page 2617.

Section 28 Solutions

Section 28.1

28.1.1. Even though the result is `X'0000000C'` (or `0000000+`), it's unlikely any packed decimal operand would be generated this way. (Suppose the second operand of the AHI instruction had been 2140: would you expect the result at **PWord** to be a valid packed decimal number?) (Is it?) (If yes, what is its value?)

Section 28.2

28.2.1. Decimal operands may have differing lengths, whereas both operands in a binary addition or subtraction have the same length (perhaps after internal sign extension). If the decimal operands have the same length, then no decimal overflow will occur when adding numbers of unlike sign.

28.2.2. The results are:

```
(1) X'04294967295C', CC=2, no overflow.
(2) X'00001C', CC=3, overflow
(3) X'99997D', CC=1, no overflow
(4) X'4D', CC=3, overflow
(5) X'111C', CC=2, no overflow
(6) X'0C', CC=0, no overflow
(7) X'000D', CC=3, overflow
(8) X'7D', CC=3, overflow
```

28.2.3. The results are:

```
(1) X'12690C', CC=2, no overflow.
(2) X'000D', CC=3, overflow
(3) X'000C', CC=0, no overflow
(4) X'458C', CC=3, overflow
(5) X'040000000D', CC=1, no overflow
(6) X'00000C', CC=0, no overflow
```

Section 28.3

28.3.1. The subtraction would overflow, setting the Condition Code incorrectly to 3 instead of 2, and possibly also causing an interruption for decimal overflow.

28.3.2. The CC will be set to 1, because the first operand (`X'0C'`) is logically less than the second (`X'0D'`). Remember that the Assembler generates the preferred sign codes!

28.3.3. The CC settings are:

```
(a) CC=1
(b) CC=2
(c) CC=1
(d) CC=1
(e) CC=2
(f) CC=0
(g) CC=1
(h) CC=1
```

28.3.4. The CC settings are:

```
(a) CC=2
(b) CC=1
(c) CC=0
(d) CC=0
(e) CC=1
(f) CC=0
```

28.3.5. The operands may overlap only if their rightmost bytes coincide.

28.3.6. You can't use CP to compare a zoned decimal value, because the `X'F'` zone digit causes a Data Exception.

Section 28.4

28.4.2. Let the first and second operands be N1 and N2 bytes long respectively. Then operand 2 (the multiplier) can have at most $2 \times N2-1$ nonzero digits, and operand 1 (the multiplicand) is required to have at most $2 \times (N1-N2)-1$ nonzero digits. (Remember that $N1>N2$.) The largest number of nonzero digits in the product is the sum of these, or $2 \times N1-2$, but the product is $2 \times N1-1$ digits long.

28.4.3. Because the product is at most 16 bytes long, the *shorter* operand must be at most 8 bytes long. Even though multiplication is commutative (a*b is the same as b*a), the fact that operand 1 is to receive the product forces operand 2 to be the shorter operand.

28.4.4. The largest valid first and second operands are

```
Operand1 DC    PL16'999999999999999'   (15 9's)
Operand2 DC    PL8'999999999999999'    (15 9's)
```

so the largest product is `X'0999 9999 9999 9998 0000 0000 0000 001C'`. (Spaces added for readability.)

28.4.5. The products are shown in this table:

	Original Pair	Product	Reversed Pair	Product
(a)	(9+) × (9−)	X'081D'	(9−) × (9+)	X'081D'
(b)	(72−) × (7+)	X'00504D'	(7+) × (72−)	X'00504D'
(c)	(44+) × (44+)	X'0001936C'	(44+) × (44+)	X'0001936C'
(d)	(15−) × (55+)	X'0000825D'	(55+) × (15−)	X'0000825D'
(e)	(107+) × (107+)	X'0011449C'	(107+) × (107+)	X'0011449C'
(f)	(28+) × (3−)	X'00084D'	(3−) × (28+)	X'00084D'

28.4.6. The result will not be 063+. Even though the first operand field is long enough to hold the product, it does not have as many leading zero bytes as the length of the second operand. This will lead to a Data exception, and the IC will be set to 7.

Section 28.5

28.5.2.

(1) 113+055+
(2) decimal divide interruption
(3) 67957045711−1+
(4) 520322163+202+
(5) 38460−018+

28.5.3. In 08−35− the quotient sign digit is not in the low-order position of the second byte, and the remainder doesn't have the same length as the divisor. (Another clue: packed decimal numbers always have an odd number of digits.)

Section 28.7

28.7.4. It seems simplest to add positive and negative terms separately, and then add those two results. But if the list is long, the number of digits needed to hold the intermediate sums increases, and the addition time for each term therefore increases also. The choice probably depends on operand sizes and the number of items in the list. (There's no simple answer, unfortunately.)

Section 29 Solutions

Section 29.1

29.1.1. The CC settings are:

```
(1)    CC=3
(2)    CC=0
(3)    CC=1
(4)    CC=2: the operand is X'00FF'
(5)    The operand is invalid because it is more than 16 bytes long.
       (You can't represent pi to this many digits as a packed decimal
       constant.)
(6)    CC=0: the operand is X'879696845A', or 879696845+
```

29.1.2. Not really. You could start with the rightmost byte and step the starting address of the operand to the left one byte at a time; but if any byte had two invalid digits, you couldn't tell which it was. You might get better results with a TRT instruction and an appropriate translate table.

Also, stepping from right to left would also require remembering which byte had the most recent bad digit.

29.1.3. Because valid digits are in the range from 0 to 9, you could use this test:

```
         CLI   BadByte,X'9F'
         JH    BadLeft            c(BadByte) > X'9F'
         J     BadRight           c(BadByte) < X'9F'
```

This test fails if the byte is actually valid. (Why?)

Section 29.2

29.2.1. The second bytes containing the two length digits for the 7 ZAP instructions are respectively X'10', X'10', X'12', X'12', X'11', X'10', and X'11'.

29.2.2. When (1) a preferred sign digit is desired, (2) when validity checking of sign and digits is desired, or (3) to set the CC to reflect the sign of the operand.

29.2.3.

```
(1) C(X) = X'123D', CC = 1;
(2) C(X) = X'405C', CC = 2;
(3) C(X) = X'000C', CC = 0;
(4) C(X) = X'753C', CC = 2;
(5) data exception, caused by invalid digit;
(6) data exception, caused by invalid sign.
```

29.2.4. It might be tempting to try

```
         XI   PackVal+L'PackVal-1,1
```

However, this won't work if the sign code is E or F, both of which indicate +.) One way is to construct a translate table which leaves the left half of the byte unchanged, and which changes the right half to a preferred sign code of the opposite sense. One way is this:

```
         ZAP   Temp,PackVal       Copy to a temporary
         ZAP   PackVal,=P'0'      Clear the original field
         SP    PackVal,Temp       Copy back with opposite sign
         - - -
Temp     DS    PL(L'PackVal)      Temporary, same size as PackVal
```

An even simpler way to do this (courtesy of John Ganci) is:

```
         ZAP   PackVal,PackVal         Make a preferred sign (C or D)
         XI    PackVal+L'PackVal-1,1   Invert the sign
```

29.2.5. These are two possible disadvantages; there may be others!

- You must remember to check for operand fields of equal length. Because MVC has only a single length field, it might be easy to change the length of one of the operands and forget that the MVC statement will not be correct after the program is re-assembled.
- MVC doesn't set the Condition Code, and does not check for invalid results.

29.2.6. The maximum value of the Length Expression in the MVC instruction is 256, so we must have

```
NDec-1 = 256/L'Dec
```

which (with L'Dec equal to 3) gives NDec=86. (Why is there a −1 in the above expression?)*

29.2.7. These two instructions will do the job:

```
ZAP   SomeVal,SomeVal              Force preferred sign code
NI    SomeVal+L'SomeVal-1,X'FE'    Set low-order bit to zero
```

The ZAP instruction will not change the value of the operand, but will ensure that the sign code is one of the preferred values, X'C' or X'D'. The NI instruction sets the low-order bit of the sign code to zero, making the sign code +. Note that

```
OI    SomeVal+L'SomeVal-1,X'0F'
```

sets the sign to +, but X'F' isn't the preferred plus sign. (This can be important if you're using CLC to compare lots of packed decimal data: two equal values like X'1C' and X'1F' could compare as unequal.)

You might be able to simplify the instructions, as well as create the preferred X'C' preferred sign code, starting with the OI instruction, by writing

```
OI    SomeVal+L'SomeVal-1,X'0F'    Force + sign code
NI    SomeVal+L'SomeVal-1,X'FC'    Make preferred + code
```

Section 29.3

29.3.1. The results will be

```
at A:  X'005C'      After adding X'3F' to X'002F'
at B:  X'3F'        The operation code for UNPK
at C:  X'F0F5FCF3'  The unpacked value (?!)
```

Revising the code is left as another exercise for you.

29.3.2. The overflow will occur on the 127th addition.

This result is not obvious. Consider the first several additions: the results will be 368+, 376+, 382+, 384+, 388+, etc. You can see that the low-order digit takes values 4, 8, 6, and 2 in cycles that increase the original value by 20. When the value arrives at 984+, we will have performed (984-364)/20×4=124 additions; the next two additions give 988+ and 996+, and the next addition will overflow, leaving 002+ as the result. Thus, there are 124 additions to get to 984+, and 3 more additions cause the overflow.

Don't just calculate (1000-364)/4=159, assuming an increment of 4 each time!

29.3.3. XC may be useless if a valid sign is required.

29.3.4. The only change occurs after the first addition, which produces 370+ and a CC setting of 2. Further additions cause no changes, because the rightmost byte at **XX** contains 0+. (Don't try this in a program; it might loop indefinitely!)

29.3.5. Yes. Consider (9−)−(999−): the result is 0+, the Condition Code is 3, and a decimal overflow has occurred.

29.3.6. Suppose the length of the first operand **XX** is N bytes, and that we want to zero the rightmost D digits, where D is an odd number. (Why an *odd* number?) Then the SP instruction of Figure 293 on page 502 can be written

```
SP    XX(N),XX+(N-1)-D/2((D+1)/2)
```

To do this with an NC instruction, we might write

```
         NC    XX+(N-1)-D/2((D+1)/2),Mask
         - - -
Mask     DC    (D/2)X'00',X'0F'   Leaves sign intact
N        Equ   (an odd number satisfying 0 < D < 2*N)
```

29.3.7. The results are:

Case 1: Decimal Overflow, CC=3, c(A)=X'5C'.
Case 2: CC=2, c(B)=X'075C'

Note that the result in both cases has the preferred plus sign code.

* Because the first element is already initialized before the MVC is executed.

29.3.8. A decimal overflow can only be caused by packed decimal instructions, all of which are 6 bytes long. Thus, the Instruction Length code cannot be 1.

29.3.9. A 4-byte packed decimal number can hold 7 digits, while each element of the Dec array is at most 5 digits long. Thus, adding 50 elements can't overflow.

Section 29.4

29.4.1. To test for valid signs and digits, or to set the CC to zero.

29.4.2. The last. (Why?)

29.4.3. With the CP instruction, the CC setting will be 0, because positive and negative zeros are equal. With the CLC instruction, the CC will be 1 because X'000C' < X'000D'. A different choice of sign codes can change the results!

29.4.4. These instructions don't use a temporary variable; the address in GR2 points to the current maximum value.

```
NDec     Equ   50                    Number of table entries
         LA    0,NDec-1              Initialize count to (#entries-1)
         LA    2,Dec                 Current largest
         LA    1,Dec+L'Dec           Start compare with second entry
Compare  CP    0(L'Dec,2),0(L'Dec,1) Compare current max to element
         JH    Next                  Branch if max is bigger
         LR    2,1                   Save address of new max value
Next     LA    1,L'Dec(,1)           Step pointer to next element
         JCT   0,Compare             Count and loop
         ST    2,BigItemA            Save address of max value
         - - -
BigItemA DS    A                     Address of largest element
Dec      DS    (NDec)PL3             Table of elements
```

How would you modify these instructions to set GR3 to the *index* of the largest element (for example, to 1 for the first element, 2 for the second, etc.)?

29.4.5. Remember that the hex representation of the two literals: =P'+10' is X'010C', and =P'-20' is X'020D'. The CP instruction will set the CC to 2 (the first operand is greater), but the CLC instruction will set the CC to 1 (the first operand is lower). This shows why you must know the data types when choosing instructions to compare operands.

29.4.6. No. You might believe that the ZAP instruction would validate the new value, but that would have happened already during execution of the CP instruction. (But MVC might be more efficient.)

29.4.7. The operands are X'12' and X'70'; neither is signed.

Section 29.5

29.5.1. $2 \leq N_1 \leq 16$, $1 \leq N_2 \leq 8$, $N_1 > N_2$. (That was easy, wasn't it!)

29.5.2. Eight, because it is used as the length of the multiplier in the MP instruction, and because 2*LD is used in the length expression for several other operands.

29.5.4. An MP instruction with identical operands will create a specification exception, because L_1 is equal to instead of greater than L_2.

29.5.5. If the multiplier is a single digit, as in

    007+ × 7+

and we assume that the multiplier digit 7+ is fetched once, then the result should be the expected 049+. However, if the multiplier is longer than one byte, as in

    0000777+ × 077+

the result would be unpredictable, depending on the order in which multiplicand and multiplier bytes are fetched. In general, it's best to avoid overlapping operands in decimal multiplication.

29.5.6. X'049C'.

29.5.7. X'0003969C'.

29.5.8. X'07151577489C'.

29.5.9. The contents of A will be unchanged; the Assembler will complain that the first operand of the MP instruction is not longer than the second.

Section 29.6

29.6.1.If we assume that the preferred sign codes (X'C' and X'D') are used, then the test for a zero element could be written

```
Test     CP    0(ELen,1),=P'0'    Check for zero magnitude
         JNE   AddUp              Okay to add
         TM    ELen-1(1),X'01'    Test rightmost sign bit
         JO    Divide             Finished (almost) if it's 1
```

If the other four sign codes are used, we must modify the tests to also check for a sign code of X'F':

```
Test     CP    0(ELen,1),=P'0'    Check for zero magnitude
         JNE   AddUp              Okay to add
         TM    ELen-1(1),X'01'    Test rightmost sign bit
         JZ    Addup              Not negative; OK to add
         TM    ELen-1(1),X'0F'    Check for + sign with X'F'
         JNO   Divide             Branch if not +0 to divide
```

29.6.2. $2 \le N_1 \le 16$, $1 \le N_2 \le 8$, and $N_1 > N_2$.

29.6.3. A DP instruction with identical operands will cause a specification error, because L_1 is equal to L_2 instead of being greater.

29.6.4. You could do a decimal divide by 2 and test the remainder; or, you could use a TM instruction like this:

```
         TM    PData+L'PData-1,X'10'   Test rightmost decimal digit
         JZ    Even                    Branch if it's even
         - - -
PData    DC    P'1234567'              Sample data
```

29.6.5. The condition code setting for a +0 and a −0 is always zero, so you can't know what the sign is.

29.6.6. This is guaranteed to produce a specification exception, because L_1 will necessarily be less than or equal to L_2.

29.6.7. If the divisor is one digit long, as in

```
         00000067+  ÷  7+
```

then the result might be the expected 00009+4+. However, if the divisor is longer than a single digit, it is difficult to predict the result. In general, it's best not to use overlapping operands in decimal division.

29.6.8. X'00009C4C' (quotient 9, remainder 4).

29.6.9. X'00001C000C'. (quotient 1, remainder 0)

29.6.10. X'00016C495C'. (quotient 16, remainder 495)

29.6.11. There is no *obvious* reason why this should be the case; you can easily visualize dividing a 31-digit dividend by a 25-digit divisor to give a 25-digit remainder and a 5-digit quotient. The answer may be that (a) limited CPU internal space for long operands, since division by a 25-digit divisor could involve many repeated subtractions. By limiting the divisor to 15 digits and sign, the internal circuits would not have to carry more than 8 bytes at a time.

29.6.12. Review the discussion at the end of Section 28.5.

29.6.14. The result is X'82479C030C047C', with quotient 82479 and remainder 30. (The fact that the divisor is adjacent to the dividend and divisor is only meant to make the problem a little more confusing.)

29.6.15. Number of quotient digits = $2 \times (\text{dividend_length} - \text{divisor_length}) - 1$

Section 29.7

29.7.1. Both expressions result in an unsigned displacement X'FFFFFFFE'. This is then larger than 4095, the largest possible value for the displacement D_2. The statement should be written

```
         SRP  A,64-2,0            Shift 2 digits to the right
```

She could also specify a USING statement like

```
         USING 5,-2
```

but (as with all similar promises to the Assembler) would have to be sure that GR5 contains −2 when the SRP instruction is executed. (Try it, and see what the assembled instruction looks like!)

29.7.2. Because at least one high-order zero digit must have been introduced, any carry generated by the rounding process cannot propagate past the end of the shifted operand.

29.7.3. If the rounding digit is added in binary, a carry might not propagate into the next higher-order decimal digit of the first operand, or an invalid digit might be generated.

29.7.4. Both instructions will cause decimal data exceptions.

29.7.5. The results are

1. 998+ (no shift!)
2. 950+
3. 010+

29.7.6. The shift amount is −1, a one-digit right shift, and the resulting data will be X'00457D' = PL3'-457'.

29.7.7. The shift operand (63) is the same as −1, so a single right shift with rounding will be done. The result will be X'00457D', not X'00456D', because the rounding digit is given the same sign as the shifted operand.

29.7.9. Feel free to stop when you've written 100 statements in your attempted solution. Congratulate yourself if you can find a solution (including CC settings) less than 100 statements long. Now you know why SRP was invented!

29.7.10. Using a signed shift amount meant that only one shift instruction was needed, rather than two, one for left shifts and one for right shifts. Also, the maximum length of a packed decimal operand is 31 digits (16 bytes), while binary shifts can handle up to 63 bits (8 bytes).

29.7.12. Our programmer friend forgot that the rounding digit is (internally) given the same sign as the decimal operand. The Assembler will complain that the rounding operand digit I_3 is not between 0 and 9.

29.7.13. The result is c(E)=X'500C' with decimal overflow causing CC3. Note the preferred plus sign.

29.7.16.

1. If the left shift causes no decimal overflow, and
2. shifting the operand left does not cause a decimal overflow exception, then
3. the method will give a rounded quotient (with the usual slight bias).

29.7.17. Consider these: (1) add 5- to 5-; the (overflowed) result will be 0-. (2) multiply 0+ by 0-. (3) divide 000+ by 1-. (4) use SRP to shift all significant digits of (say) -5 off the right end of the field.

Section 29.8

29.8.2. The results are

1. X'00000123456C'
2. X'000C1C2C3C4C'
3. X'0000001E240C'
4. X'23456789ABCC'

29.8.3. The number of *numeric* digits moved is always even; the number of hex digits moved (including the sign) is always odd.

29.8.4. The number of numeric digits truncated is always odd.

29.8.6. The possible movements for the digits, and an instruction that does the movement, are shown below:

```
1L → 2L              MVZ   Byte2,Byte1
1R → 2R              MVN   Byte2,Byte1
1R → 2L              MVO   Byte2,Byte1
1L → 2L, 1R → 2R     MVC   Byte2,Byte1
1L → 2R, 1R → 2L     PACK  Byte2,Byte1 (OR UNPK)
1L → 2R              Let me know how you did it.
```

29.8.7. First, the operation of the instructions:

- The two MVC instructions create an 8-byte field at Out with copies of the byte.

- The NC instruction masks off all but the single bit at each of the bit offsets from zero to seven. For example, the first four bytes will contain X'80 40 00 10'.
- The MVO instruction moves the first four bytes to the Work field, offset to the right by 4 bits (effectively right-shifting).
- The following MVC moves the four right-shifted bytes back to their original positions at Out.
- The TR instruction converts each bit value to an EBDCIC 0 or 1 character.

The literal must be nine characters long, because the possible offsets into the string (from the bit values at Out) can take values from 0 to 8.

29.8.8. (a) $(N_2-N_1)\times2+1$. (b) $(N_1-N_2)\times2-1$.

Section 29.9

29.9.1. The operand 0001234+ has only one byte of high-order zeros, but the operand 100+ is two bytes long.

29.9.2. The most important restriction is that the result operand must be 16 or fewer bytes long; this can be written

```
L + N/2 < 16 .
```

Similarly, the original operand must be 15 or fewer bytes long, so L < 16 and N > 0. The required instruction sequence then takes the form

```
         MVN   A+L+N/2(1),A+L-1
         MVO   A(L+1),A(L)
         NC    A+L(N/2+1),=XL(N/2+1)'F'
```

29.9.3. We require L < 16, N > 1, and N < 2L-1. Then we can use the code sequence

```
         MVN    B+L-1(1),A+L-1    Move sign
         MVO    B(L-N/2),A        Move digits
         NC     B+L-N/2-1(N/2+1),=XL(N/2+1)'F'
         - - -
A        DC     PL(L)'initial value'
B        DS     PL(L)             Shifted result goes here
```

29.9.4. We require L ≤ 16 and 0 < N < 2L. The MVN instruction can then be written

```
         MVN   A+L-N/2-1(1),A+L-1
         - - -
B        DS    0PL(L-N/2)         Define length attribute of result
A        DS    PL(L)              Original L-byte operand
```

29.9.5. Suppose N = 2L−1, which means that all the digits of the operand are to be "shifted off". Then the MVO instruction

```
         MVO   A,A(0)
```

might appear to do the job. However, as you remember from the discussion of the length digits in two-length SS-type instructions, a zero in the Length Expression is assembled as a zero length digit in the instruction. This is equivalent to

```
         MVO   A,A(1)
```

which performs a shift of N−2, or 2L−3, digits, rather than N as required.

29.9.6. These instructions illustrate a possible solution.

```
         LA    1,Over             Set address of overflow branch
         CLI   A,X'09'            Check for high-order zero digit
         JNH   Okay               Skip if digit is zero
         SR    1,1                Reset branch address
Okay     MVO   A(L),A(L)          Perform the single shift
         NI    A+L-1,X'F'         Reset the low-order digit to zero
         LTR   1,1                Now test the overflow switch
         BNZR  1                  Branch to Over if an overflow
```

29.9.7. We require L ≤ 16 and 0 < N < 2L. The following instructions perform the required shift.

```
         MVC   A(L-N/2),A+N/2          Shift left N digits
         NC    A+L-N/2-1(N/2+1),Mask   Mask out vacated digits
         - - -
Mask     DC    X'F0',(N/2-1)X'0',X'0F'
```

29.9.8. The test for lost significant digits might be written

```
         LA    1,Over            Set branch address
         OC    A(N/2),A          Check n high-order digits
         JNZ   Over              Branch if not all zeros, overflows
         SR    1,1               Set branch flag for no overflow
BadNews  DC    0H
         - - -                   Shift as in previous solution
         LTR   1,1               Check overflow flag
         BNZR  1                 Branch if something was lost
```

29.9.9. Because the result must be 16 or fewer bytes long, we must have $L+N/2 \le 16$; in addition, $N > 0$ and $0 < L \le 15$. The instructions can then be written

```
         MVC   A+L+N/2-1(1),A+L-1 Move sign
         NC    A+L-1(N/2+1),Mask  Zero intervening digits
         - - -
Mask     DC    X'F0',(N/2-1)X'0',X'0F'
```

29.9.10. Consider the following instructions:

```
         MVC   A+L'A-1+N/2(1),A+L'A-1
         NC    A+L'A-1(N/2+1),Mask
         - - -
Mask     DC    X'F0',(N/2-1)X'0',X'0F'
A        DC    PL(length)'value'
         DS    PL(N/2)
```

The length of **A**, plus N/2, should be less than or equal to 16; otherwise the result will be too long to be used as a valid packed decimal operand. The value of N must be even and greater than zero.

29.9.12. We require $L \le 16$ and $N \le 2\times(L-1)$.

Section 29.10

29.10.1. The values are:

- A   (3,0)
- B   (3,3)
- Sum (7,0)

Section 29.11

29.11.1. The values are:

1. P'1024.2048'   I'=5, S'=4
2. P'-0.98765'   I'=2, S'=5
3. P'+2235058.4' I'=8, S'=1
4. P'72.3456'   I'=3, S'=4

Programming Problem 29.2.

This little program shows the result. The first two executable instructions set the decimal overflow bit in the Program Mask to zero, to disable the interruption for decimal overflow.

```
P29_2    CSect ,
         Using *,15              Establish addressability
         Print NoGen             Don't generate the macro expansions
         XR    1,1               Set GR1 to zero
         SPM   1                 Set decimal overflow interrupt off
         AP    M5,M5             Add (5-) and (5-)
         Printout M5,*,Header=No Print the result and stop
M5       DC    P'-5'             Single-byte 5-
         End   P29_2
```

Note that CC=3 is displayed on the Printout message, as expected.

Programming Problem 29.3.

This little program shows the result.

```
P29_3    CSect ,
         Using *,15                Establish addressability
         Print NoGen               Don't generate the macro expansions
         MP    P2,M0               Multiply (2+) and (0-)
         Printout P2,*,Header=NO   Print the result and stop
M0       DC    P'-0'               Single-byte 0-
P2       DC    PL2'2'              Two-byte 002+
         End   P28_3
```

Section 30 Solutions

Section 30.1

30.1.1. Twelve. The largest 64-bit signed magnitude is 2^{63} = 9,223,372,036,854,775,808 (19 digits), and the 16-byte receiving field can handle 31 digits.

30.1.2. An 8-byte field holding a packed decimal number has room for 16 hex digits, of which 15 are numeric decimal digits and the last is a sign code.

30.1.3. c(X) = X'000002147483648D'. (That should have been easy!)

30.1.4. The first character string printed would be Page 0002I, with a letter I where the "9" should be. This is caused by the UNPK instruction leaving the sign code digit "C" as the zone of the low-order byte of the page number. When the page number is 30, the field will print as "Page 0003?", where the rightmost character position indicated by ? is an unprintable character having representation X'C0'. (What will actually appear on the printed page depends on the behavior of your printers and your Operating System.) When the page number is 31, Page 0003A will be printed.

30.1.5. Here's one way to do it:

```
         - - -
         OI    ZonePgn+NDigits-1,C'0'  Set zone of final digit
         LA    0,NDigits-1             Set digit count less 1
         LTR   0,0                     Test if only 1 digit wanted
         JNP   Finish                  Exit if so
         LA    1,PageNo+L'PageNo-NDigits    Address of first digit
Blot     CLI   0(1),C'0'               Check for leading zero
         JNE   Finish                  Exit loop if nonzero
         MVI   0(1),C' '               Set to blank
         LA    1,1(,1)                 Step to next digit position
         JCT   0,Blot                  Count down and loop
Finish   - - -
```

If we knew that the digit count specified by **NDigits** would always be greater than 1, we could omit the test for a single digit. Note that we always leave one visible digit, even if the page number is zero.

30.1.6. The values are

1. X'000000000000001D'
2. X'000002147483321C'
3. X'000002004318083C'
4. X'000001077952604C'

30.1.7. The values are

1. X'0000000000000000000000000000001D'
2. X'0000000000009223372036854775807C'
3. X'0000000000008262604149049818200C'
   (Be careful! the argument value has 17 hex digits, so the high-order digit was lost.)
4. X'0000000000006717250228839017560D'

Section 30.2

30.2.1. A fixed-point divide exception will occur, because the packed decimal operand is $+2^{31}$. The result in the first operand register will be X'80000000'.

30.2.2. The instructions will execute correctly. GR0 will contain X'0000007B', and the contents of **DWork** will be X'000000000000123C'.

30.2.3. The Assembler will issue an error message for the PACK instruction, because the (implied) length of the second operand exceeds 16 bytes!

30.2.4. Converting a blank data field is a common error in using CVB. After the PACK instruction is executed, the contents of **WorkArea** will be X'0000000000000004', which has an invalid sign code. Thus, the CVB instruction will cause a program interruption for a data exception, and the IC will be set to 7.

30.2.5. Because $\log_2(10^{15}) = 49.83$, at least 50 binary digits would be needed to compute the magnitude correctly. Since the high-order bits will be lost even if the result is complemented, 50 bits would be sufficient.

One way to do the conversion is by successive multiplications and additions; *no* divisions need be done! If the partial terms of the intermediate value are multiplied by 10 and the new low-order digit is added, only 32 bits need to be carried, but overflows must be detected so that an invalid result can be indicated.[335] It would be inadvisable to indicate a fixed-point overflow condition, since the program mask can be set to ignore them, and the program would have to check the CC for an overflow indication; but CVB doesn't set the CC!

30.2.7. The low-order 32 bits of the result might be useful if the remainder (mod 2^{32}) has some significance to the program. But to use the result, the program would first have to handle the program interruption.

30.2.8. The resulting values are:

1. `X'0098967F'`
2. `X'D3A8C00E'`
3. `X'FFFFFFFF'`
   (Note that the high-order digit of the DC statement operand was truncated!)
4. `X'000003E7'`

Section 30.3

30.3.1. The patterns are:

1. FC is a blank; the pattern is `C'•dddddsd'`.
2. FC = `C'*'`; the pattern is `C'*dddsd.ss'`. The only Message Character is the period before the last two Digit Selectors.
3. FC = `C'*'`; the pattern is `C'*•dddsdf•Total=dddsd.dd'`. The Message Characters are the blanks before and after the first set of selectors, the characters `C'•Total='`, and the period before the last two Digit Selectors.
4. FC is a blank; all other bytes are Message Characters.

30.3.2. The pattern is represented by `C' '''('CR)'(''`. One problem with reading this pattern is that each interior apostrophe represents a single byte, and they aren't paired; however, if they're paired, then it's even harder to tell what the pattern is!

Section 30.4

30.4.1. The result at **`LineX`** will be C'••123456789•', because the final digit of the decimal number will not be accessed.

30.4.2. The Fill Character *can* be a digit selector; thus L−1 digits, and therefore as many as (L−1)/2+1 bytes, might be taken from the second operand. If there are no digit selectors in the pattern, no bytes would be taken from the second operand.

30.4.3. The results will be

1. `C'••••••729413'`, where • represents a blank space.
2. `C'**0000729413'`
3. `C'*****0729413'`

30.4.4. The results will be

1. `C'••••••••7294'`, where • represents a blank space.
2. `C'**0000007294'`
3. `C'*****0007294'`

30.4.5. Because the SS character in the pattern immediately precedes the last DS of the pattern, eleven blanks followed by a single zero digit will replace the pattern. The DS marks the position of the last leading zero to be suppressed.

30.4.6. Because there are at most 10 significant digits in the decimal representation of the value contained in a fullword, the first digit of the 6-byte (11-digit) packed decimal number is always zero. Thus the first DS is always replaced by the Fill Character.

30.4.7. Since *no* decimal digits will be sought at the second operand address, no exception condition due to the invalid address will be recognized.

[335] It is strange that an error in a multiplicative process can be indicated by a divide exception!

Section 30.5

30.5.1. Be careful! *Pattern* characters may affect the Significance Indicator, but no *Message* Character affects the SI.

30.5.2. The SS character must appear one byte earlier, so the pattern becomes `C' dd,dsd.dd•CREDIT'`:

```
Pat2     DC    C' ',X'20206B2021204B2020',C' CREDIT'  Pattern
```

If the amount was 0000002+, the edited result would be "••••••0.02•CREDIT".

30.5.3. There should be 2×P−1 d and s selectors.

30.5.4. A packed decimal operand P bytes long contains 2×P−1 digits. Also, to produce any useful results at all, we must have D < P. For example, if D=P−1, only a single digit can be formatted.

1. Even if the desired number of digits N is even, there must be an odd number of selectors: 2×(P−D)−1. The leftmost digit of the packed decimal operand must be zero if an even number of digits is to be displayed correctly.
2. To display N digits requires B=[(N+2)/2] bytes, where the square brackets mean that the quotient should be rounded down to the next lower integer. Thus, the offset D is P−B. (In mathematical notation, we should write B=floor((N+2)/2), where floor(X) is the largest integer less than or equal to X.)

30.5.5. If c(Num) = 0, only a single zero digit will be printed; all the preceding Message Characters are insignificant. If c(Num) = 512, no commas will appear in the result because the SI is turned on by the first significant digit ("5"), which follows the last comma.

30.5.6. The number to be edited has 10 significant digits, and the Fill Character in the pattern is a blank. Remember that a packed decimal number must have an odd number of digits, so the second operand of the ED instruction has a leading zero that is suppressed by the pattern.

30.5.7. The result would be `C'•••••••.00•'`.

30.5.8. The result would be `C'•••••••.00•CREDIT'`.

Section 30.6

30.6.1. This code fragment shows one way this can be done:

```
         MVC   Display,Pattern    Move pattern to print line
         LA    1,Display+L'Display-1   Point to possibly forced digit
         EDMK  Display,PackVal    Edit and mark up to 7 digits
         JNZ   DoSign             If not zero, set sign
         XR    2,2                Clear GR2
         IC    2,PackVal+L'PackVal-1   Insert rightmost byte
         NILL  2,X'000F'          Clear all but sign digit
         IC    2,SignBits-X'A'(2) Get sign indicator
         LTR   2,2                Set CC (0 means -, 1 means +)
DoSign   BCTR  1,0                Adjust to preceding byte
DoPlus   MVI   0(1),C'+'          Assume result was +
         JP    Print              Branch if the guess was right
DoMinus  MVI   0(1),C'-'          Negative result, set - sign
Print    PrintLin  Display,L'Display   Print result
         - - -
SignBits DC    AL1(1,0,1,0,1,1)   Sign indicators +,-,+,-,+,+
PackVal  DC    PL4'0'             Packed decimal operand
Pattern  DC    C' ',5X'20',X'2120'    Pattern
Display  DS    CL8
```

Note that the instruction sequence

```
         XR    2,2                     Clear GR2
         IC    2,PackVal+L'PackVal-1   Insert rightmost byte
         NILL  2,X'000F'               Clear all but sign digit
```

could be replaced by

```
         IC    2,PackVal+L'PackVal-1   Insert rightmost byte
         NILF  2,X'0000000F'           Clear all but sign digit
```

30.6.2. The CC setting will be zero, because all the digits of the result are zero. It is possibly misleading that the SI will still be ON at the end of the operation, due to the − sign code in the right half of the last source byte. The result of the edit is C'AA00'.

30.6.7.

```
(1) For argument P'7',P'2468', the result is C'•7•2••468'
(2) For argument X'67891234',  the result is C'•678••912'
```

30.6.8. The result is `C'••12••34••56789'`

Section 30.7

30.7.1. If each of the second operand bytes contains a sign code in the right digit, N bytes will be taken from the second operand. If none of the first N digits of the second operand is a sign code, as few as (N+1)/2 bytes might be taken from the second operand. (Remember, the Field Selector FS doesn't cause any second operand access.)

30.7.2. The CPU retains the digit it has, and doesn't reset the second-operand "digit fetcher" to get a new left-hand digit.

30.7.3. GR1 will point to a character in the last source field *only* if there are significant nonzero digits stored into that field, and significance was not forced by a SS character in the pattern.

Section 30.8

30.8.1. The Significance Indicator is set OFF as follows:

- at the start of the operation,
- after a Field Separator (FS) pattern character, and
- after a source byte containing a + sign code in the right-hand digit.

The Significance Indicator is set ON as follows:

- pattern = SS, source = any nonterminal digit;
- pattern = DS, source = nonzero terminal digit;
- pattern = SS, source = terminal digit with − sign code;
- pattern = DS, source = nonzero terminal digit with − sign code.

Programming Problem 30.3.

This program scans for multiple data values on each input record.

```
P30_3    CSect ,                      Problem 30.3
         Print NoGen                  Don't show macro expansions
         Using *,15                   Establish addressability
         PrintLin Title,L'Title       Print a title line
         LA    10,10                  Initialize multiplier
*
GetRec   ReadCard InBuff,EndFile
         MVI   OutLine,C' '
         MVC   OutLine+1(L'Outline-1),OutLine   Initialize output line
         LA    2,InBuff               Point to start of record
         LA    3,L'InBuff-1           Set effective length of record
Scan     EX    3,ScanTRT              Scan over leading blanks
         JZ    GetRec                 No more data on this record
         LR    7,1                    Save pointer to initial digit
         XR    5,5                    Initialize accumulated value
GetDigit CLI   0(1),C' '              Check for terminating blank
         JE    Display                Go display this result
         IC    6,0(,1)                Insert a numeric character
         N     6,=XL4'F'              Mask all but numeric digit
         MR    4,10                   Multiply partial value by 10
         ALR   5,6                    Add new digit
         BC    3,TooBig               Carry out means overflow
         LR    8,4                    Copy accumulated value
         LR    9,5                    ...both halves
         SLDA  8,32                   Check for overflow
         JO    TooBig                 Value is too large
         LA    1,1(,1)                Step to next character
         J     GetDigit               And get additional digits
*
Display  CVD   5,Work8                Convert value to packed decimal
```

```
         MVC   OutVal,ValPatt          Move pattern to output area
         ED    OutVal,Work8+2          Convert to edited characters
         J     MovChars                Go move the input characters
*
TooBig   MVC   OutVal,=CL12'**OverFlow**'   Insert overflow message
StepOver CLI   0(1),C' '               Check for end of digits
         JE    MovChars                Reached the end, go move them
         LA    1,1(,1)                 Step to next character
         J     StepOver                Continue looking
*
MovChars LR    8,1                     Copy pointer to ending blank
         SR    8,7                     Subtract string start address
         BCTR  8,0                     Make an effective length
         EX    8,OutMVC                Move the character string
         PrintLin  OutLine,L'OutLine   Display the results
         MVC   OutLine+1(L'OutLine-1),OutLine    Clear the print line
*
         LR    2,1                     Point where rescan starts
         LA    3,InBuff+L'InBuff-1     Point to end of record
         SR    3,2                     Remaining length to scan
         BCTR  3,0                     Make it an effective length
         J     Scan                    Look for another digit string
EndFile  PrintOut *,Header=NO          End the program
*
ScanTRT  TRT   0(*-*,2),SkipBlnk       Skip blanks
OutMVC   MVC   OutChars(*-*),0(7)      Move character string
ValPatt  DC    C' ',9X'20',X'2120'     Edit pattern
Title    DC    C' Converted Value    Input characters'
OutLine  DC    CL100' '                Output line
OutVal   Equ   OutLine+4,12            Position/length of value
OutChars Equ   Outline+20              Position of input characters
SkipBlnk DC    (C' ')X'1',X'0',(255-C' ')X'1'    Translate table
InBuff   DS    CL80                    Buffer for input records
Work8    DS    CL8                     CVD work area
         End   P31_3
```

The sample data was in these records:

```
 1
 0002    0003
 1234567890    2147483647
 9999999999    000000000000000000000042
```

and the output was these records:

```
 Converted Value    Input characters
              1    1
              2    0002
              3    0003
     1234567890    1234567890
     2147483647    2147483647
   **OverFlow**    9999999999
             42    000000000000000000000042
```

Programming Problem 30.7. This solution uses a loop to eliminate leading zeros from the unpacked value. (Compare it to your solution for Problem 17.4.)

```
P30_7    CSect ,
         Using P30_7,15           Provide addressability.
* Set up initial values.
         LA    2,0                Previous value F(0).
         LA    3,1                Current value F(1).
         PrintLin Title,L'Title
         PrintLin OutRec,1
*
Loop     CVD   3,WrkArea          Convert to packed decimal
         UNPK  Number,Wrkarea+2(6)  Unpack to EBCDIC
         OI    Number+L'Number-1,X'F0'  Set correct zone on last digit
         LA    5,Number           Start of output value
Check    CLI   0(5),C'0'          Is it a leading zero?
         BNE   PrintNum           If not, print the result
         MVI   0(5),C' '          If yes, replace it with a blank
         LA    5,1(5)             Step to next character
         B     Check              And try the next digit
PrintNum PrintLin OutRec,OutLeng  Print the result
         LR    4,2                Get a copy of the previous value.
         AR    4,3                Compute the next value.
         BO    Done               Overflow means we're done.
         LR    2,3                Copy current value to previous.
         LR    3,4                Copy next value to current.
         B     Loop               Play it again, Sam.
*
DONE     PrintLin EndRec,L'EndRec Print an ending line.
         BR    14                 Return to caller.
*
WrkArea  DS    D
OutRec   DC    C' '               Carriage control for output.
Number   DS    CL12               Output from conversion
OutLeng  EQU   *-OUTREC           Length of record.
*
Title    DC   C'1Fibonacci Sequence to Maximum Positive Fullword Value'
EndRec   DC    C'0Program ends.'
         END   P30_7
```

The printed output looks like this:

```
1Fibonacci Sequence to Maximum Positive Fullword Value

           1
           1
           2
           3
   ..........
    701408733
   1134903170
   1836311903
0Program ends.
```

Programming Problem 30.8. This solution uses the ED instruction to format the result with commas separating each group of three digits. (Compare it to your solution for Problem 30.7.)

```
* Print a Fibonacci Sequence up to Maximum Fullword Value.
P30_8    CSect ,
         Using P30_8,15           Provide addressability
*
* Set up initial values.
*
         LA    2,0                Previous value F(0).
         LA    3,1                Current value F(1).
         PrintLin Title,L'Title
         PrintLin OutRec,1
*
Loop     CVD   3,WorkArea         Convert to packed decimal
         UNPK  Number,Workarea+2(6)  Unpack to EBCDIC
         MVC   OutRec(OutLeng),EdPat Move edit pattern to output area
         ED    OutRec(OutLeng),WorkArea+2  Format the result
         PrintLin OutRec,OutLeng  Print the result
         LR    4,2                Get a copy of the previous value.
         AR    4,3                Compute the next value.
         BO    Done               Overflow means we're done.
         LR    2,3                Copy current value to previous.
         LR    3,4                Copy next value to current.
         B     Loop               Play it again, Sam.
*
Done     PrintLin EndRec,L'EndRec Print an ending line.
         BR    14                 Return to caller.
*
WorkArea DS    D
EdPat    DC    C' '               Fill character
         DC    X'20206B2020206B2020206B202120' dd,ddd,ddd,dsd
OutRec   DC    C' '               Carriage control for output.
Number   DS    CL15               Output from ED
OutLeng  EQU   *-OutRec           Length of record.
*
Title    DC   C'1Fibonacci Sequence to Maximum Positive Fullword Value'
EndRec   DC    C'0Program ends.'
         END   P30_8
```

The printed output looks like this:

```
1Fibonacci Sequence to Maximum Positive Fullword Value

             1
             1
             2
  .............
           987
         1,597
  .............
     3,524,578
  .............
   701,408,733
  .............
 1,134,903,170
 1,836,311,903
0Program ends.
```

Section 31 Solutions

Section 31.2

31.2.1.

1. 0.1 (base 10) = $.0001\underline{1001}$ (base 2), X'1$\underline{9}$' (base 16).
2. 0.3142 (base 10) = 0.0221110011 (base 3) = 0.458244 (base 14).
3. X'BBBBB...' = 0.733333333... (base 10) = 0.AEEEEE... (base 15).
4. 3.6 (base 8) = 3.75 (base 10) = X'3C'.

31.2.2.

1. X'0.DEFACE' = 0.87101448
2. X'0.5' = 0.3125
3. X'0.C2854' = 0.75984573
4. X'0.333333' = 0.19999999
5. X'0.BEEF' = 0.74583435

31.2.3. These are the four hexadecimal fractions:

1. X'0.10504816F...'
2. X'0.00068DB8BAC7...'
3. X'0.FCD38CDA...'
4. X'0.00003C53E2D6...'

31.2.4. (1/3) base 2 = .0101010101...

31.2.5. The values are shown, where underscored groups of digits are repeated indefinitely.

$0.1_{10} = 0.0\underline{6314}$
$0.2_{10} = 0.\underline{1463}$
$0.3_{10} = 0.2\underline{3146}$
$0.4_{10} = 0.\underline{3146}$
$0.5_{10} = 0.4$
$0.6_{10} = 0.\underline{4631}$
$0.7_{10} = 0.5\underline{4631}$
$0.8_{10} = 0.\underline{6314}$
$0.9_{10} = 0.7\underline{1463}$

Section 31.3

31.3.1. The values could be defined as follows:

```
Amount   DC    F'1174949'         Amount in cents (scaled by 10**2)
Percent  DC    F'5147'            Percentage rate (scaled by 10**3)
Round    DC    F'0.5E5'           Rounding factor
Correct  DC    F'1E5'             Correction factor
```

31.3.2. The generated constant is X'0E8D4A51'.

31.3.3. The two constants will generate X'1174949C' (4 bytes) and X'05147C' (3 bytes) respectively. The product work area must therefore be at least 7 bytes long.

31.3.4. The hexadecimal values are:

1. X'6.5F5C28F5...'
2. X'3DB.9A5E353...'
3. X'13A.28C59E0...'

31.3.5. With signed 32-bit words, you can create 32 different representations, and with unsigned words, 33. This may seem surprising, but remember that the radix point can be placed to the left of the leftmost significant digit as well as to the right of the rightmost digit.

31.3.6. You need 10 bits for the integer part ($2^9<604<2^{10}$). The fraction part isn't finite (or at least, appears not to be!).

31.3.7. The hexadecimal constant is X'3243F6A9'.

```
         DC    FS(-11)'10E11'   = 10**12 = X'1D1A94A2'
         DC    FS(-12)'U10E12'  = 10**13 = X'9184E72A'

         DC    FDS(-25)'U10E25' = 10**26 = X'295BE96E64066972'
         DC    FDS(-26)'U10E26' = 10**27 = X'CECB8F27F4200F3A'
```

31.3.9. This way of writing the constant seems natural, as it looks like "Ten to the 12th power". But it's actually 10 followed by 12 zeros, which is too large for a signed constant.

31.3.10. This solution is partially parameterized using the symbols SF and SD:

```
X31_3_10 CSect ,
         Using *,15
SF       Equ   24                Binary Scale Factor
SD       Equ   5                 Significant fraction digits
         L     1,A               Get value to convert
         LR    0,1               Copy to GR0
         SRL   1,SF              Discard fraction digits
         CVD   1,D               Convert integer portion
         UNPK  Int,D             Unpack to EBCDIC
         OI    Int+L'Int-1,X'F0' Correct low-order zone digit
         CLI   Int,C'0'          Is hign-order digit zero?
         JNE   DoFrac            If not, continue
         MVI   Int,C' '          Set the zero to space
DoFrac   DC    0H                Now do the fraction part
         L     1,A               Get value to convert
         SLL   1,32-SF           Eliminate integer portion
         SRL   1,32-SF           Reposition fraction digits
         M     0,=FE(SD)'1'      Multiply to 10**SD (significance)
         SRDL  0,SF-1            Drop all but rounding bit
         AHI   1,1               Add 1 to round
         SRL   1,1               Drop off rounding bit
         CVD   1,D               Convert to packed decimal
         UNPK  Frac,D            Unpack fraction digits
         OI    Frac+L'Frac-1,X'F0'     Set correct zone digit
         Printout Result,*,Header=No   Print result
A        DC    FS(SF)'98.234567' Value to convert
D        DS    D                 CVD conversion area
Result   DS    0CL(3+1+SD)
Int      DC    CL3' '            Integer portion
         DC    C'.'              Decimal point
Frac     DC    CL(SD)' '         Fraction digits
         End   X31_3_10
```

Section 31.4

31.4.1. Avogadro's Number is $6.022\times10^{+23}$, and Planck's constant is 6.626×10^{-34}.

31.4.2. Avogadro's Number is slightly less than 10^{24}, and Planck's constant is slightly greater than 10^{-35}, so you would need enough bits to represent 10^{59}, or about 200 bits. (Now you can see why floating-point is useful for some types of applications!)

Section 31.5

31.5.1. Zero will have representations with a zero fraction, and 19 possible exponent values from −9 to +9, including zero. Usually the preferred representation of zero will use a zero exponent, so that the other 18 are redundant.

31.5.2. There are six possible representations, three with the significant digits represented as fractions and three with the significant digits represented as integers.

```
Fractions:    .0047  e=+3    .0470  e=+2    .4700  e=+1
Integers:      0047. e=-1     0470. e=-2     4700. e=-3
```

Section 31.7

31.7.1. The original implementation of floating-point registers on System/360 provided only registers 0, 2, 4, and 6, so it was natural to pair them as (0,2) and (4,6). When the remaining 12 registers were added, it was important to retain the original pairings (so that existing programs would continue to execute correctly); the other pairings were made to follow the same pairing style.

31.7.2. The CPU need only OR the given register digit with B'0010'.

Section 31.8

31.8.1. DD, EB, L or LH, LD.

Section 31.9

31.9.1. Here are three possibilities: the first requires two storage references and a doubleword in memory; the second uses FPR6 as a temporary, and the third uses GGR1 as a temporary.

```
         STD   0,DTemp          LDR   6,0          LGDR  1,0
         LDR   0,4              LDR   0,4          LDR   0,4
         LD    4,DTemp          LDR   4,6          LDGR  4,1
         - - -
DTemp    DS    D
```

You can create other (more elaborate) ways to do this, such as

```
         LDGR  1,0          Copy FPR0 to GG1
         LDGR  2,4          Copy FPR4 to GG2
         XGR   1,2          Do the
         XGR   2,1          .. XOR
         XGR   1,2          .. swap
         LGDR  0,2          Copy original contents of FPR4 to FPR0
         LGDR  4,1          Copy original contents of FPR0 to FPR4
```

but no reasonable programmer would do it this way.*

31.9.2. Use the floating-point registers as intermediate “storage”.

31.9.3. LXY and STXY for single load and store, and perhaps LMF and STMF (or LMFP and STMFP) for muultiple load and store. The most important exception condition would be to check that the register operand is a valid reference to an extended register pair.

Programming Problem 31.1.

This solution uses packed decimal arithmetic:

```
P31_1    CSect ,
         Using *,15
         Print NoGen
         ZAP   PDProd,PDAmt       Copy amount to work area
         MP    PDProd,PDTax       Multiply by tax rate
         SRP   PDProd,64-5,5      Convert to cents and round
         MVC   PDAns,PDPat        Move pattern to output area
         LA    1,PBAns+7          Point just past possible $ sign
         EDMK  PDAns,PDProd+3     Edit the result
         BCTR  1,0                Move pointer left 1 byte
         MVI   0(1),C'$'          Insert dollar sign
         PrintOut PDAns,*,Header=NO    Print result
PDAmt    DC    P'11749.49'        Amount to be taxed
PDTax    DC    P'.05147'          Tax rate
PDProd   DS    PL8                Work area for product
PDPat    DC    C' ',X'4020202020202021204B2020' Formatting pattern
PDAns    DC    CL12' '            Output area for result
         End   P31_1
```

The printed result will look like this:

```
 PDAns    = '     $604.75'
```

* Some people think Assembler Language programmers are inherently unreasonable.

Programming Problem 31.2.

This solution uses scaled fixed-point binary arithmetic:

```
P31_2    CSect ,
         Using *,15
         Print NoGen
         L     1,FBAmt             Put argument in GR1
         M     0,FBTax             Multiply by tax rate
         AL    1,FBRnd             Add rounding factor
         JC    12,FBNoC            Skip if no carry
         AHI   1,1                 Propagate carry
FBNoC    D     0,FBConv            Correct to get result in cents
         CVD   1,FBWork            Convert to decimal
         MVC   FBAns,FBPat         Move pattern to output area
         LA    1,FBAns+7           Point just past possible $ sign
         EDMK  FBAns,FBWork+3      Edit, mark first significant digit
         BCTR  1,0                 Move pointer left 1 byte
         MVI   0(1),C'$'           Insert dollar sign
         PrintOut FBAns,*,Header=NO    Print result
FBAmt    DC    F'1174949'          Rate in cents
FBTax    DC    F'5147'             Tax rate per 100,000 cents
FBRnd    DC    F'0.5E5'            Round to nearest cent
FBConv   DC    F'1E5'              Convert result to cents
FBWork   DC    D'0'                Integer part in packed decimal
FBPat    DC    C' ',X'4020202020202021204B2020' Formatting pattern
FBAns    DC    CL12' '             Output area for result
         End   P31_2
```

The printed result will look like this:

```
 FBAns    = '     $604.75'
```

Programming Problem 31.3.

This isn't an easy problem, but it shows the efforts that programmers made when only fixed-point binary arithmetic was available.

```
P31_3    CSect ,
         Using *,15
         Print NoGen
SF       Equ   28                  Scale factor
FBLoop   L     0,FBA               Put argument in GR0
         SRDA  0,32-SF             Shift right, now scaled by 2**28
         D     0,FBX               Divide by current scaled estimate
         S     1,FBX               Subtract it to get the correction
         SRA   1,1                 Divide by 2
         LPR   2,1                 Copy magnitude for tolerance test
         A     1,FBX               Add original estimate
         ST    1,FBX               Store new value
         LPR   2,2                 Take magnitude of correction
         CHI   2,3                 Check correction smaller than 3
         JH    FBLoop              Repeat if not converged yet
         LR    4,1                 Copy for formatting
         SRDA  4,SF                Split integer and fraction parts
         CVD   4,DFBW              Convert integer part to decimal
         UNPK  FBAns(1),DFBW+7(1)  Move to answer string
         OI    FBAns,X'F0'         Put correct zone
         MVI   FBAns+1,C'.'        Place the decimal point
         SRL   5,32-SF             Reposition fraction part
         M     4,FBConv            Convert 9 fraction digits
         SRDL  4,SF-1              Position as integer value
         AHI   5,1                 Add 1 for rounding
         SRDL  4,1                 Final shift
         CVD   5,DFBW              Convert to integer
```

```
         OI    DFBW+7,X'0F'       Set correct zone
         UNPK  FBAns+2(9),DFBW+3(5)  Move fraction part to output area
         PrintOut FBAns,*,Header=NO
FBA      DC    FS(SF)'2'          Value whose root is to be found
FBX      DC    FS(SF)'1'          Initial estimate
FBConv   DC    F'1E9'             For converting fraction part
DFBW     DC    D'0'               Work area for CVD results
FBAns    DC    CL12' '            Output area for result
         End   P31_3
```

The printed result will look like this:

```
 FBAns    = '1.414213563 '
```

Section 32 Solutions

Section 32.1

32.1.1. The three values are:

1. 0.9450×10^3 ($0.7654\times0.1235=0.09452690$)
2. 0.5859×10^4 ($0.7655\times0.7655=0.58599025$)
3. 0.1000×10^1 ($0.0333\times0.0321=0.00106893$)

Section 32.2

32.2.1. A nonzero operand can have at most $p-1$ leading zero digits, so at most $p-1$ shifts are required.

32.2.2. The two products are:

Not pre-normalized	Pre-normalized
0.1000×10^2	0.1029×10^2
Zero!	0.5183×10^1

Section 32.3

32.3.1. The (possibly unexpected) results are the following:

(1) .9995 for each iteration
(2) 1.001, 1.002, 1.003, 1.004
(3) 1.000 for each iteration

Section 32.4

32.4.1. Any operand of the form .xxxD with digit $D\neq0$. Its exponent doesn't matter.

32.4.2. The values are shown in this table:

Product	No guard, no round	Guard, no round	Guard and round
155×165	$.2550\times10^5$	$.2557\times10^5$	$.2558\times10^5$
45×2469	$.1111\times10^6$	$.1111\times10^6$	$.1111\times10^6$
21×1117	$.2340\times10^5$	$.2345\times10^5$	$.2346\times10^5$
127×137	$.1730\times10^5$	$.1739\times10^5$	$.1740\times10^5$

32.4.3. The values are shown in this table:

Product	No guard, no round	Guard, no round	Guard and round
509×101	$.5140\times10^5$	$.5140\times10^5$	$.5141\times10^5$
509×555	$.2824\times10^6$	$.2824\times10^6$	$.2825\times10^6$
509×150	$.2790\times10^6$	$.2799\times10^6$	$.2800\times10^6$
407×515	$.2090\times10^6$	$.2096\times10^6$	$.2096\times10^6$

32.4.4. With only a guard digit, the results are these:

1. 0.9452×10^3
2. 0.5859×10^4
3. 0.1068×10^1

With both guard and rounding digits, the results are these:

1. 0.9453×10^3
2. 0.5860×10^4
3. 0.1069×10^1

32.4.6. (a) The product without a guard digit will be 0.0560 so the test will indicate that it is less than 0.0562. (b) With a guard digit, the product will be 0.0565, which is greater than 0.0562.

Section 32.5

32.5.1. One advantage is that steps of the multiplication process can stop when the remaining multiplier or multiplicand digits are known to be zero; it can also avoid the necessity of right-shifting the product digits if the product would be longer with rightmost zeros.

For example, $(1230.\times10^0)\times(4500.\times10^0)$ or $1230\times4500 = 5535000$ has 3 low-order zeros; in FPI(10,4) three corrective right shifts would be needed. If internal arithmetic used only 4 digits, it's possible some high-order digits could be lost, or an error condition might be indicated.

Section 32.6

32.6.1. The shift requirements are shown in this table:

Operation	Quotient	Shifts	Rounded quotient
.1428 ÷ .7142	.2000	0	.2000
.6667 ÷ .6666	1.00015	1	.1000
.1277 ÷ .3456	.3695	0	.3695
.3456 ÷ .1275	2.71059	1	.2711
.9999 ÷ .9999	1.0000	1	.1000

If the dividend ("numerator") fraction is greater than or equal to the divisor ("denominator") fraction, the quotient will be greater than 1, requiring a shift.

Section 32.7

32.7.1. The calculated results are shown in this table:

Operation	No guard, no round	Guard, no round	Guard and round
$(.7435\times10^3)-(.9621\times10^1)$	$.7338\times10^3$	$.7338\times10^3$	$.7339\times10^3$
$(.7435\times10^1)-(.6994\times10^1)$	$.4410\times10^0$	$.4410\times10^0$	$.4410\times10^0$
$(.1043\times10^5)-(.9527\times10^4)$	$.9000\times10^3$	$.9030\times10^3$	$.9030\times10^3$
$(.1000\times10^0)-(.9992\times10^{-2})$	$.9000\times10^{-1}$	$.9000\times10^{-1}$	$.9001\times10^{-1}$

32.7.2. If you subtract zero, the magnitude of the other operand doesn't decrease.

Section 32.8

32.8.1. The relative sizes of the largest ulps are these:

Representation	Max relative ulp
FPF(16,14)	$\approx 16^{-13}$
FPF(2,24)	$\approx 2^{-23}$
FPF(10,34)	$\approx 10^{-33}$

Section 32.9

32.9.1. Consider these three pairs:[336]

1. [+9‖+.2000] ÷ [−9‖+.4000] generates [+18‖+.5000]
2. [−9‖+.2000] ÷ [+9‖+.4000] generates [−18‖+.5000]
3. [−9‖+.2000] ÷ [+2‖+.4000] generates [−11‖+.5000],

   which if denormalized becomes [−9‖+.0050],

32.9.2. These are the results:

1. [+9‖+.1038] (Normal)
2. [+7‖−.4830] (Normal)

336 You could just provide the examples shown in Section 32.9 (with keys **1**, **2**, and **5**) but that would be less interesting.

Section 32.10

32.10.1. Max=[+63‖X'FFFFFF'], which with bias=X'7FFFFFFF'; and Min=[−64‖X'100000'] which with bias=X'00100000'.

32.10.2. Because Min is approximately 16^{-65}, its reciprocal is about 16^{+65}. Since Max is approximately 16^{+63}, the range asymmetry means that there are about $p \times 16^{2}$ more small values than large, where “p” is the precision of the fraction.

Section 33 Solutions

Section 33.1

33.1.1. Five possible interpretations, and their assembler language defining statements, are shown in these statements:

```
DC    F'1077952604'    Fullword binary integer
DC    C'•••*'          Characters (three blanks and an asterisk)
STH   4,X'05C'(0,4)    Store halfword
DC    P'+4040405'      Packed decimal
DC    E'.250982046'    Hexadecimal floating-point
```

Other interpretations are possible.

33.1.2. For short hexadecimal floating-point numbers, (a) 5, (b) 6, (c) 6, (d) 256. For long numbers, (a) 13, (b) 14, (c) 14, (d) 256. (The reason there are 256 redundant values for zero is that pseudo-zeros of either sign are included.)

33.1.3. The values are (1) Normalized, (2) Normalized, (3) Unnormalized, (4) Zero, (5) Normalized, (6) Pseudo-zero, (7) Normalized.

33.1.4. The 14-bit characteristic would have a range from -2^{13} to $+2^{13}-1$, or approximately (−8192, +8191), assuming the bias is chosen to provide range symmetry.

Section 33.2

33.2.1. The easiest way to tell is to assemble a long constant also:

```
DC    E'1E13'  generates X'4B9184E7'
DC    D'1E13'  generates X'4B9184E72A000000'
```

So the short constant does not contain all the significant bits. (You could also check the table of the hexadecimal representation of powers of ten in the Appendix.)

33.2.2. This is one of many possibilities:

```
DC    E'1,1e1,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9,1e10'
```

33.2.3. The tables in the Appendix showing the hexadecimal representations of powers of 10 are helpful!

1. Short: 10^9 = X'3B9ACA00' (or X'483B9AC' in short HFP format) has six nonzero significant digits.
2. Long: 10^{22} = X'21E19E0C9BAB2400000' (or X'5321E19E0C9BAB24' in long HFP format) has fourteen nonzero significant digits.
3. Extended: 10^{46} = X'1C06A5EC5433C60DDAA16406f5A400000000000' (or X'671C06A5EC5433C6 590DDAA16406F5A4' in extended HFP format), has 28 nonzero significant digits.

33.2.4. The nine values are:

```
4019999999999999A       DC    D'0.1'
4033333333333333        DC    D'0.2'
404CCCCCCCCCCCCD        DC    D'0.3'
4066666666666666        DC    D'0.4'
4080000000000000        DC    D'0.5'
409999999999999A        DC    D'0.6'
40B3333333333333        DC    D'0.7'
40CCCCCCCCCCCCCD        DC    D'0.8'
40E6666666666666        DC    D'0.9'
```

Section 33.3

33.3.1. For the constants named A and D, only a single byte is available to hold the sign and characteristic, leaving no room for the fraction! So, the Assembler will complain about lost precision. For the other constants, the generated data is

```
B        X'401A'        The fraction is rounded to 8 bits
C        X'433E'        999=X'3E7', and the 7 is truncated
E        X'433F'        1000=X'3E8', and the 8 causes rounding up
```

33.3.2. These are the values and their generated constants:

W	3.7×10^3	X'43E7400000000000'
X	1.0×10^{55}	X'6E6867A6'
Y	8.8×10^0	X'418CCCCCCCCCCCCC 33CCCCCCCCCCCCCD'
Z	1.0×10^{18}	X'503782DB'

Section 33.4

33.4.2. With half-even rounding, the L-type constant generates X'1E177FF880000000 1000000000000000' so you can't tell whether the rounding digit (the last 8) should be rounded up (if there are any nonzero bits in the distance) or not (that is, to the even final digit 8). So, the Assembler must calculate the value of the constant to *more* than 112 bits. (Some nonzero bits *do* appear with additional precision, so the Assembler knows the value should be rounded up, as shown.)

33.4.3. The rounding modifiers and the generated values are:

R1 X'3E4189374BC6A7F0'
R4 X'3E4189374BC6A7F0'
R5 X'3E4189374BC6A7EF'

33.4.4.Here's an example using the last of the values in the table: when the constant's value is evaluated to a precision of 30 hexadecimal digits, the result is
584F341F 25338E9D 527E3486 4A16 | 7C
where the final digit of the generated constant (6) will not be rounded up because the next 8 bits are B'01111100'.

When the precision is increased to 38 hexadecimal digits, the result is
584F341F 25338E9D 527E3486 4A16 | 80000000 03
where the final rounding bits indicate that the generated constant should be rounded up.

Section 33.6

33.6.1. The result in FPR(0,2) will be

```
FPR0: X'50123456789ABCDE'  (unchanged)
FPR2: X'42DCBA9876543210'  (corrected characteristic and sign)
```

and the CC will be 2 indicating a positive result.

33.6.2. These are the CC settings and register contents:

```
(1) CC=2, c(FPR4) = X'4C428571'
(2) CC=1, c(FPR2) = X'CC42857196DBB933'
(3) CC=1, c(FPR4,6) = X'8A123456789ABCDE FC42857196DBB933'
(4) CC=2, c(FPR4) = X'4C42857196DBB933'
```

Section 33.7

33.7.1. The primary change is to use MER rather than MEER.

```
         XR    5,5                GR5 contains index for XX and YY
         XR    6,6                GR6 contains index for ZZ
         LZDR  4,4                Set long FPR4 to zero
         LH    2,Count            Counter in GR2
Loop     LE    0,XX(5)            Load X(i) in FPR0 (short)
         LE    4,YY(5)            Load Y(i) in FPR4 (now, long)
         MER   0,0                Long X(i)*X(i) in FPR0
         MDR   0,4                Multiply by Y(i)
         STD   0,ZZ(6)            Store long result Z(i)
         LA    5,L'XX(,5)         Increment XX, YY index by 4
         LA    6,L'ZZ(,6)         Increment XX index by 8
         JCT   2,Loop             Count down and loop
         - - -
XX       DC    E'1,2,3,4,5,6,7,8,9,10' Values of X(i)
Count    DC    Y((*-XX)/L'XX)     Count of X(i) entries
YY       DC    5E'3.14159,2.71828'     Values of Y(i)
ZZ       DS    10D                Long results Z(i)
```

33.7.2. The resulting fraction has only 12 significant digits, and the two low-order fraction digits are always zero.

33.7.3. They could be different. ME generates an 8-byte product that needs no rounding, because the low-order byte of the product is X'00'. MEE generates only a 4-byte product, for which a rounded result would depend on the leftmost bit of the right half of the long product.

33.7.4. Here's one way to calculate the table of cubes:

```
        LA    0,100             Set count of results wanted
        LE    2,=E'1'           Constant 1 in FPR2
        LER   6,2               Initialize N
        LA    1,Cubes           Set GR1 to start of table
CubeIt  LER   4,6               Copy N to FPR4
        MER   4,4               Form N**2
        MER   4,6               Form N**3
        STE   4,0(,1)           Store in the table of cubes
        LA    1,L'Cubes(,1)     Increment table address
        AER   6,2               Increment N by 1
        JCT   0,CubeIt          Repeat for 100 values
        - - -
Cubes   DS    100E
```

33.7.5. c(FPR0)=`X'4112345699999999'`; the right half of FPR0 is unchanged.

33.7.6. This leads to the peculiar (and possibly disturbing) circumstance that for a large class of hexadecimal floating-point numbers (those with a nonzero low-order digit), multiplication by 1 would not yield the original operand!

33.7.7. Simply XOR `X'40` to the sum.

Section 33.8

33.8.1. The Assembler generates the rounded constant `E'0.6' = X'4099999A'`.

33.8.2. The results are `X'3F9ABCD8'` and `X'3E9ABC80'` respectively.

33.8.3.

1. c(FPR0)=`X'427D93482D335B9E'`.
2. c(FPR0)=`X'C53BA1816A7D80C7'`.

33.8.4. An exponent underflow interruption will occur if the Program Mask bit is one, and c(FPR4)=`x'7F100000'`. If the Program Mask bit is zero, c(FPR4)=0, and no interruption occurs.

33.8.5. HER and HDR can be thought of as analogs of

```
        SRA   x,1
and
        SRDA  x,1
```

Section 33.9

33.9.1. The result is X'5812345698765400'.

33.9.2. The instruction will generate a specification exception, because both operands don't refer to the lower-numbered register of a floating-point register pair. FPR2 will be unchanged.

33.9.3. The mnemonic for “Add Unnormalized” is AU, and the mnemonic for “Add Double Unnormalized” is AW, or “A Double U”.*

33.9.4. The results are:

```
1.  c(FPR0)=X'40130000', CC=2
2.  c(FPR0)=X'15400000', with an exponent overflow interruption
3.  c(FPR0)=X'45000000', CC=0
4.  c(FPR0)=X'C2831693', CC unchanged
```

33.9.5. The results are:

* Engineers find humor wherever they can.

1. c(FPR0)=X'42FE00F0', CC=2
2. c(FPR0)=X'41104F3D', CC=2
3. c(FPR0)=X'42800078', CC unchanged
4. c(FPR0)=X'00222242', with an exponent overflow interruption
5. c(FPR0)=X'46384000', with an exponent underflow interruption
6. c(FPR0)=X'C117D861', CC=1

33.9.6. The results are:

1. c(FPR0)=X'C1200000', CC=1
2. c(FPR0)=X'44002000', CC=2
3. c(FPR0)=X'44000000', with an HFP lost-significance interruption
4. c(FPR0)=X'04000000', with an HFP lost-significance interruption
5. c(FPR0)=X'0B012335', CC=2
6. c(FPR0)=X'4601579A', CC=2

33.9.7. The AER instruction will cause an interruption for exponent overflow, and the result in FPR0 is X'801FFFFF'.

33.9.8. Consider this addition:

```
         LE    2,=X'7FFFFFFF'
         AE    2,=X'7E0FFFFF'
```

Because the characteristic difference is one, the second operand must be right-shifted one digit, so that the CPU adds the fractions X'FFFFFF' and X'00FFFF', giving the sum X'1.00FFFFE', which after a normalizing right shift gives the result characteristic X'100', indicating an exponent overflow and a final characteristic of zero. Any smaller second operand would produce a smaller sum.

33.9.9. This operation subtracts $(1-16^{-6})$ from 1, so the expected result is $+16^{-6}$.

1. SE produces c(FPR0) = X'3B100000', or 16^{-6} as expected.

The result of the unnormalized subtraction may be surprising:

2. SU produces c(FPR0) = X'41000000', with a significance exception. To see how this happens:

```
      41100000(0)   becomes   41000000(0)
     -40FFFFFF(0)            -410FFFFF(F)
      3B100000(0)             40000000(1)
```

   Remember, the guard digit does *not* participate in the test for a zero intermediate fraction.

33.9.10. None.

33.9.12. Because the first operand is 10^6 and the second is 10^{-6}, their sum must be able to represent at least 12 decimal digits. This means that either long or extended hexadecimal floating-point operands and arithmetic must be used.

33.9.13. The result is X'57234569'; the guard digit (the high-order digit “9” of the smaller operand) appears in the result.

33.9.14. The results are:

(1) X'41110005'
(2) X'4080004B'
(3) X'40FFFFFF'
(4) X'41100000'
(5) X'40223456'

Section 33.11

33.11.1. The result would have been X'3FDCF140', showing that an additional digit of precision is generated.

33.11.3. X'BB100000'

33.11.4. The cases where unequal operands may compare equal include:

1. normalized compared to unnormalized
2. unnormalized compared to unnormalized
3. pseudo-zero compared to pseudo-zero

Here are some examples:

```
000000 7800 F0B0            000B0     5        LE    0,=x'42123000'  Normalized
000004 7820 F0B4            000B4     6        le    2,=x'44001230'  Normalized
000008 3902                           7        cer   0,2
000044 7920 F0B8            000B8   758        ce    2,=x'43012300'  Unnormalized
000078 7840 F0BC            000BC   774        le    4,=x'75000000'  Pseudo-zero
00007C 7940 F0C0            000C0   775        ce    4,=x'05000000'  Pseudo-zero
```

In all cases, the comparison Condition Code is zero.

Section 33.12

33.12.1. The fraction of the X'3B10wxyz' operand must be shifted right five digits for operand alignment, giving the internal subtraction

```
  40000001(0)
 -40000001(0)
  40000000(0)
```

Because the guard digit is included for comparisons, the two operands appear to be equal, and the Condition Code is set to zero. (Try it!)

33.12.2. These are the resulting Condition Code settings:

1. (1) CC = 1
2. (2) CC = 2
3. (3) CC = 0
4. (4) CC = 2

33.12.5. Only if one or both operands is unnormalized, or both are zero or pseudo-zero. Try comparing these operands:

```
(1)  X'42123000'  to  X'44001230'  norm to unnorm
(2)  X'44001230'  to  X'43012300'  both unnorms
(3)  X'75000000'  to  X'03000000'  both pseudo-zero
```

Section 33.13

33.13.1. Since the carry that caused the overflow was the result of adding a low-order 1-bit, the result fraction must be X'.1000...'. Because the characteristic "wraps around" from 127 to 0, the result must be either X'00100...' or X'80100...'.

33.13.2. The three results are:

1. c(FPR0)=X'0010000087654321', rounded, with an interruption for exponent overflow.
2. c(FPR0)=X'4000000187654321' (rounded).
3. c(FPR0)=X'4000000012345678' (not rounded).

33.13.3. The sequence works. For example, if FPR2 is initialized to X'4200000080000000', the rounded result stored at **Round3** is X'42100001'.

33.13.4. There will be no difference. The LDER instruction extends the short number from **Variable** with zero, so the LEDR instruction cannot round the result in FPR0.

Section 33.14

33.14.2. The largest positive integer is X'7FFFFFFFFFFFFF00' = +9223372036854775552, and the result is X'50FFFFFFFFFFFFFF'. The largest negative value is X'8000000000000000' = −9223372036854775808, and its result is X'D080000000000000'.

The magnitude of the negative number is 256 larger than the magnitude of the positive number because not all 63 bits of the positive number can be held in the 14 hex digits of the long hexadecimal floating-point result.

33.14.3. No, because the largest 64-bit integer is less than 10^{19}, well within the range of all three hexadecimal floating-point formats.

33.14.4. The first instruction changes the sign bit of the integer operand. Remember that the two's complement representation of a 32-bit binary integer X is $(2^{32}+X)$ (modulo 2^{32}). Thus, if the integer was originally positive,

$$(2^{32} - X) - 2^{31} = 2^{31} - X$$

is stored. This quantity is placed in FPR0 and 2^{31} is subtracted from it. The result is the normalized value of X.

33.14.8. All three can be generated as hexadecimal floating-point D-type constants, but the first causes an Assembler error:

```
000008 4E00000000000000    DCon1 DC DS(14)'0.1'             Error, lost significance
** ASMA073E Precision lost
000018 CE00000080000000    DCon2 DC DS(6)'-2147483648'      -2**31, unnormalized
000028 4F08000000000000    DCon3 DC DS(1)'36028797018963968' +2**55, unnormalized
```

The obscurity of these constants shows why a hexadecimal constant can be simplest.

33.14.12. The statement is

```
Float    DC    DS6'2147483648'
```

33.14.13. This is one way to do it:

```
         L     0,IntVal           Get the integer
         ST    0,T+4              Store in pseudo-zero
         LZDR  6                  Set FPR6 to zero
         AD    6,T                Create the normalized HFP value
         STD   6,W                Store the result for testing
         TM    W+4,X'80'          Is the 'lost' bit 1?
         JZ    OK                 If not, no rounding needed
         XC    W+1(3),W+1         Clear high-order 6 digits of result
         AD    6,W                Add the rounding bit
OK       STE   6,Result           Store the rounded
         - - -
T        DC    0D,X'4E',7X'0'     Doubleword aligned pseudo-zero
W        DC    D'0'               Doubleword temporary
```

33.14.14. This is one way to do it:

```
         L     0,IntVal           Get the integer
         ST    0,T+4              Store in pseudo-zero
         LZDR  6                  Set FPR6 to zero
         AD    6,T                Create the normalized HFP value
         STD   6,W                Store the result for testing
         CLC   W+4(4),=X'80000000'  Is it exactly halfway between?
         JNE   Round              If not, do normal round up
         TM    W+3,X'1'           Is the low-order bit zero?
         JZ    OK                 If so, result is even, don't round
Round    TM    W+4,X'80'          Is the 'lost' bit 1?
         JZ    OK                 If not, no rounding needed
         XC    W+1(3),W+1         Clear high-order 6 digits of result
         AD    6,W                Add the rounding bit
OK       STE   6,Result           Store the rounded
         - - -
T        DC    0D,X'4E',7X'0'     Doubleword aligned pseudo-zero
W        DC    D'0'               Doubleword temporary
```

33.14.15. The results are shown in this table:

HFP_Number	c(GR3)	CC	c(GG3)	CC
X'46000000'	X'00000000'	0	X'0000000000000000'	0
X'C7654321'	X'F9ABCDF0'	1	X'FFFFFFFFF9ABCDF0'	1
X'7FEDCB49'	X'7FFFFFFF'	3	X'7FFFFFFFFFFFFFFF'	3
X'ABCDEF74'	X'00000000'	1	X'0000000000000000'	1
X'4974662B'	X'7FFFFFFF'	3	X'000000074662B000'	2

33.14.16. The converted 32-bit value is X'00010001'. Unfortunately, the halfword value is then X'0001'!

33.14.17. Using the value at DFloat, the result stored at DTemp will be X'4E00000001000003', and the number at NF will be X'01000003'.

Section 33.15

33.15.1. The results in FPR0 after each step are:

- After DE: X'41140000' (1.25)
- After FIER: X'41100000' (1.0)
- After ME: X'42100000' (16.0)
- After AE: X'41400000' (4.0)

33.15.2. The AU instruction causes the fraction part of c(FPR0) to be shifted off; the ME instruction will pre-normalize the integer part in FPR0, so the results will be the same.

33.15.3. In this case, the addition could cause the guard digit to be part of the normalized result. In Exercise 33.15.2, the guard digit is not shifted left because there is no post-normalization.

If the quotient of the DE instruction is greater than or equal to X'46100000', no “fraction” digits will be shifted off.

33.15.4. This is one way to do it:

```
         LD    0,A               Load A into FPR0
         DD    0,B               Divide by B
         FIDR  0,0               Drop off the fraction part
         MD    0,B               Form product with integer part
         LCDR  0,0               Form -B*IntPart(A/B)
         AD    0,A               Add A to form remainder
         STD   0,AModB           Store result
         - - -
AModB    DS    D                 Remainder
A        DC    D'20'             A value for A
B        DC    D'16'             And for B
```

33.15.5. The results in FPR0 are:

```
(1) X'42120000'
(2) X'C7FEDCA9 80000000'
(3) X'00000000 00000000'
(4) X'77654321
(5) X'C7FEDCA9'
```

Section 33.16

33.16.1. Using short HFP arguments (the results would not change for long or extended arguments):

- Sqrt(Max) = X'60400000'
- Sqrt(Min) = X'20400000'
- Sqrt(DMin) = X'1E100000'

33.16.2. If a carry occurred, the result fraction would have to be all one-bits, or $(1-\text{ulp})$. But this would mean that the source operand would have to be $(1-(\text{ulp}/2))$, which can't happen.

33.16.3. First, remember that the argument and the result have the same length. Now, if the result were to lie exactly half way between two representable target values, it would have to be finite in length. But that would mean that for some exponent, it's an integral value, and hence its square (the source operand) would also be integral. (A number whose square root is an integer is itself an integer.) But square roots have about half as many digits as the source value; and since source and target have the same number of digits, the square root must be shorter than the source, so it can't lie exactly halfway between two representable values.

Section 33.19

33.19.1. This operation subtracts $(1-16^{-6})$ from 16, so the expected result is $15+16^{-6}$. The actual results are:

(1) SE produces c(FPR0) = X'41F00001' or $15+16^{-5}$ with an error of 15/16 ulp.

(2) SU produces c(FPR0) = X'420F0000', an unnormalized value of 15 with error 16^{-6} or 1/16 ulp.

So the unnormalized result is more accurate! This peculiarity is caused by unrounded hexadecimal floating-point arithmetic.

33.19.2. This table shows the approximate values:

Exponent width	Fraction digits	Exponent range	Fraction accuracy
7 bits	8	−64 to +63 $\approx 1.6*10^{-58}$ to $\approx 7.8*10^{56}$	$\approx 5*10^{-7}$
4 bits	9	−16 to +15 $\approx 3.6*10^{-15}$ to $\approx 3.5*10^{13}$	$\approx 6*10^{-8}$

These choices also depend on the expected uses of the representation: if customer data needs only a limited exponent range, a 4-bit characteristic might give more fraction accuracy.*

33.19.3.

```
XGR   0,0              Clear 64-bit general register
ICMH  0,8,X            Get sign/characteristic of X
OIHL  0,1              Set bit 31 to 1
LFGR  0,0              Copy X'cc000001 00000000' to FR0
DE    0,X              Divide by X
STE   0,ULPX           Store short precision ulp(X)
```

33.19.5.

```
XGR   0,0              Clear 64-bit general register
ICMH  0,8,X            Get sign/characteristic of X
OILL  0,1              Set bit 63 to 1
LFGR  0,0              Copy X'cc000000 00000001' to FR0
DD    0,X              Divide by X
STD   0,ULPX           Store long precision ulp(X)
```

33.19.7.

```
LD    4,X              Put high half of X in FR4
LD    6,X+8            Put low half of X in FR6
XGR   0,0              Clear 64-bit general register
ICMH  0,8,X            Get sign/characteristic of X
LFGR  0,0              Copy X'cc000000 00000000' to FR0
LGHI  0,1              Put 1 in GG0
LFGR  2,0              Copy X'00000000 00000001' to FR2
DXR   0,4              Divide by X
STD   0,ULPX           Store high half of ulp(X)
STD   2,ULPX+8         Store low half of ulp(X)
```

Programming Problem 33.1.

```
P33_1    START 0
         BASR  15,0             Set base register
         USING *,15             And inform the assembler
         LE    4,=E'1'          Carry 1.0 in FPR4
         LE    6,=E'-5'         And X in FPR6
Loop     STE   6,X              Store X value for printing
         LER   2,6              Set up denominator
         MER   2,2              Form X squared
         SE    2,=E'3'          Subtract 3.0
         MER   2,6              (X*X-3.0)*X
         SE    2,=E'2'          -2.0; now have denominator
         JNZ   Calc             Branch if nonzero to compute
         LE    0,Max            Otherwise set max value
         J     StoreF           And go store result
Calc     LER   0,4              Numerator = 1.0
         DER   0,2              Form F(X)
StoreF   STE   0,FX             Store value of function
         PrintOut X,FX          Print values
```

* If you are interested in the reasons the designers of System/360 chose hexadecimal instead of base-8 floating-point, you might consult this article: “An Analysis of Floating-Point Addition”, by D. W. Sweeney, IBM Systems Journal, Volume 4, Number 1, 1965.

```
         AER   6,4                 Increase X by 1.0
         CE    6,=E'+5'            Compare to last value
         JNH   Loop                Branch if not bigger to do next X
         PrintOut *,Header=NO      Terminate program
FX       DS    E                   Value of F(X)
X        DS    E                   Value of X
Max      DC    EH'(Max)'           Maximum hexadecimal floating-point value
         END   P33_1
```

The maximum hexadecimal floating-point value will be printed when X = −1 and X = +2.

Programming Problem 33.2.

This program displays the long hexadecimal floating-point values.

```
P33_2    CSect ,
         Print NoGen
         Using *,15
         LA    0,55                Set count of largest factorial
         LD    0,=D'1'             Initialize factorial value in FPR0
         LDR   2,0                 Initialize N in FPR2
         LDR   4,0                 Set increment in FPR4
         STD   0,Facts             Store 0 factorial
         LA    9,Facts+L'Facts     Set GR9 to next entry in table
FactIt   MDR   0,2                 Form F*N to form N!
         STD   0,0(,9)             Store in table
         PrintOut 32,Header=NO     Print contents of FPR0
         ADR   2,4                 Increment F
         LA    9,L'Facts(,9)       Increment table address
         JCT   0,FactIt            Repeat for remaining values
         PrintOut *,Header=NO      Terminate
Facts    DS    56D                 Storage for the factorial values
         End   P33_2
```

The largest precisely representable *short* hexadecimal floating-point value is easy to find by examining the long results to see which has nonzero digits extending past the sixth fraction digit; the last precise value is 12 factorial (12!). Finding the equivalent long value is more difficult, because the last nonzero digit of 22! (X'3CEEA4C2B3E0D80000') is X'8' and the last significant digit of 23! (X'57970CD7E2933680000') is X'6', so it's not easy to know whether the three low-order zero bits of 22! might be enough to hold the low bits of 23!, without going to higher precision.

By evaluating those two numbers in extended precision (as the hex values show), we find that 23! has a single one-bit in the 15th hex digit, so that 22! is the largest that can be stored without loss of precision in long hexadecimal floating-point format.

Programming Problem 33.3.

This solution first checks the input value for zero; otherwise, it multiplies or divides by powers of ten until the result is a fraction less than 1 but greater than 0.1, keeping track of the decimal exponent. This fraction is multiplied by 10^6, converted to integer form, and formatted for printing.

```
P33_3    CSect ,
         Print NoGen
F0       Equ   0                   Floating-point work register
         BASR  12,0                Set base register
         USING *,12                Provide addressability
Read     ReadCard InRec,Exit       Read a record
         OC    InRec(8),=8C' '     Make upper-case characters
         MVC   PHex,InRec          Move to output line for display
         TR    Inrec(8),HexCh      Translate digits to 0-F
         PACK  DWord(5),Inrec(9)   Pack into a 4-byte word
         LDE   0,DWord             Load and extend to long format
         XR    1,1                 Clear GR1 for decimal exponent
         MVI   ESign,C'+'          Assume positive result
         TM    DWord,X'80'         Check sign of input value
         JZ    PlusSign            Skip setting - sign, + is correct
         MVI   ESign,C'-'          Result is negative.
PlusSign LPER  F0,F0               Force argument sign + in register
         JZ    Zero                Fraction is zero, finish up easily
```

```
         CD    F0,=D'1'           See if argument has +/- exponent
         JE    FractOne           All done if exactly 1.0
         JH    PosExp             Branch if positive exponent
NegExp   MD    F0,=D'1.E10'       Negative exponent; increase by tens
         AHI   1,-10              And count decimal exponent down
         CD    F0,=D'1'           Where did that take us?
         JL    NegExp             If it's still too small, go again
         JE    FractOne           If exactly equal, that's nice.
PosExp   CD    F0,=D'1.E10'       Positive exponent; check size
         JL    Reduce             If in range, reduce slowly
         MD    F0,=D'1.E-10'      Multiply gives more accuracy.
         AHI   1,10               Bump decimal exponent accordingly
         J     PosExp             And do another reduction cycle
Reduce   CD    F0,=D'1'           Exponent in (+10,0); check for 10**0
         JE    FractOne           Fraction now is 1
         JL    Convert            Ready to convert if now a fraction
         MD    F0,=D'.1'          Reduce by 10**1
         AHI   1,1                Increment exponent accordingly
         J     Reduce             And go around again
Convert  MD    F0,=D'1.E6'        Convert to integer in (1,10**6)
         AD    F0,=D'.5'          Round it properly
         CD    F0,=D'1.E6'        Did it round up to 1000000?
         JNL   FractOne           It did, use fraction .100000
         CFER  0,5,F0             Convert to binary integer in GR0
         CVD   0,DWord            Convert to packed decimal
         OI    DWord+7,X'0F'      Set correct zone
         UNPK  EDigits,DWord      Place digits into string
DoExpon  DS    0H                 Convert exponent
         MVI   ExpSign,C'+'       Assume positive exponent
         LTR   1,1                Check for correct assumption
         JNM   DoExpon2           Skip if it was right
         MVI   ExpSign,C'-'       Set exponent sign -
DoExpon2 CVD   1,DWord            Convert to decimal
         OI    DWord+7,X'0F'      Set correct (positive) zone
         UNPK  Exponent,DWord     Unpack to zoned decimal
         PrintLin PLine,Exponent-PLine+2    Print result
         J     Read               Repeat for more values
Zero     MVC   EDigits,=6C'0'     Set fraction to zeros
         J     DoExpon            And go do exponent
FractOne MVC   EDigits,=C'100000' Set fraction digits to .100000
         AHI   1,1                Compensate by upping exponent
         J     DoExpon            And go do exponent
Exit     PrintOut *,Header=NO     Terminate
         LtOrg ,                  Insert literals
DWord    DC    D'0'               Work area
PLine    DC    C' X'''            Start of print line
PHex     DS    CL8                Input value
         DC    C''' = '
ESign    DS    C                  Sign of fraction
DecPt    DC    C'.'               Decimal point
EDigits  DS    CL6                Fraction digits
ExponE   DC    C'E'               Exponent indicator 'E'
ExpSign  DS    C                  Exponent sign
Exponent DS    CL2                Decimal exponent
HexCh    DC    256AL1(*-HexCh)    Define no-effect table
         Org   HexCh+C'A'         Set origin to offset of 'A'
         DC    X'FAFBFCFDFEFF'    Translate numeric digit to hex
         Org   ,                  Reset location counter
InRec    DS    CL80               Input area for records
         END   P33_3
```

Programming Problem 33.6.

```
P33_6    Csect ,
         Using *,15
HFPLoop  LD    0,HFPA             Put argument in FPR0
         DD    0,HFPX             Divide by current estimate
         SD    0,HFPX             Subtract current estimate
         HDR   0,0                Divide by 2
         LPDR  2,0                Copy for convergence test
         AD    0,HFPX             Add original estimate
         STD   0,HFPX             Store updated estimate
         CD    2,=D'1E-10'        See if correction is small enough
         JH    HFPLoop            If not, iterate again
         LDR   2,0                Save final result in FPR2
         SDR   4,4                Set FPR4 to zero for renormalization
         AW    0,HFPUnn           Unnormalize to extract integer part
         STD   0,HFPDW            Store temporarily
         ADR   0,4                Renormalize the integer part
         SDR   2,0                Remove integer part in FPR2
         L     0,HFPDW+4          Get low-order word
         CVD   0,HFPDW            Convert integer part to decimal
         OI    HFPDW+7,X'0F'      Set proper zone
         UNPK  HFPAns(1),HFPDW+7(1)    Unpack the integer part
         MVI   HFPAns+1,C'.'      Place the decimal point
         MD    2,HFPCon           Convert fraction digits to integer
         AD    2,HFPRnd           Add rounding factor
         AW    2,HFPUnn           Unnormalize to form fullword integer
         STD   2,HFPDW            Store temporarily
         L     0,HFPDW+4          Get low-order word
         CVD   0,HFPDW            Convert integer part to decimal
         OI    HFPDW+7,X'0F'      Set proper zone
         UNPK  HFPAns+2(9),HFPDW+3(5)  Move fraction part to output
         Printout HFPAns,*,Header=NO
HFPA     DC    D'2'               Argument
HFPX     DC    D'1'               Initial estimate
HFPCon   DC    D'1E9'             Conversion constant for fraction
HFPRnd   DC    D'0.5'             Constant for rounding
HFPUnn   DC    X'4E',7X'0'        Unnormalizing constant
HFPDW    DC    D'0'               Work Area
HFPAns   DC    CL12' '            Output area for result
         End   P33_6
```

The printed result is

```
 HFPAns   = C'1.414213562 '
```

Section 34 Solutions

Section 34.1

34.1.1. The three values are:

- Min = 2^{-126}
- DMax = $2^{-126}-2^{-149}$
- DMin = 2^{-149}

34.1.2. There are 23 fraction bits in a short binary floating-point number, of which we must exclude the case of all zero bits. So there are $2^{23}-1$ possible bit combinations. Allowing for either sign, there are $2\times(2^{23}-1)$ or 16,777,214 denormalized numbers.

34.1.3. The three values are:

- Short format: `X'7F800000'`
- Long format: `X'7FF0000000000000'`
- Extended format: `X'7FFF000000000000 0000000000000000'`

Section 34.2

34.2.1. The constants are shown in this table:

Format	2 Bytes	3 Bytes	4 Bytes
Short	3DCD	3DCCCD	3DCCCCCD
Long	3FBA	3FB99A	3FB9999A
Extended	3FFC	3FFB9A	3FFB999A

In the 2-byte extended format, there is no fraction, only the implied 1-bit!

34.2.2. Six possible interpretations and the corresponding assembler language defining statements can be written this way:

```
         DC    F'1077952604'    Fullword binary integer
         DC    C'•••*'          Characters (three blanks and an asterisk)
         STH   4,X'05C'(0,4)    Store halfword
         DC    P'+4040405'      Packed decimal
         DC    E'.250982046'    Hexadecimal floating-point
         DC    EB'3.0039281845' Binary floating-point
```

Other interpretations are possible, as we'll see later.

34.2.3. The types of the six values are:

1. `X'7FFFFFFF'` – QNaN
2. `X'007FFFFF'` – Denormalized
3. `X'80000000'` – −0
4. `X'00FFFFFF'` – Normal
5. `X'FF8000AB'` – QNaN
6. `X'FF800000'` – −infinity

34.2.4. You won't be happy with the results, because

- `Short` generates `X'3FB9999A'`, which has value +1.4500000 as a short operand, not 0.1!
- `ShortOne` generates `X'3FF00000'`, which has value +1.875 as a short operand, not 1!

- `Long` generates `X'3DCCCCCCCCCCCCCD'`, which has value +5.238689482212067E-11 as a long operand, not even close to 0.1!
- `LongOne` generates `X'3F80000000000000'` as a long operand, which has value +7.8125E-3, not 1!

This shows why you should be very careful when using BFP constants defined with length modifiers.

Section 34.3

34.3.1. The resulting value is X'007FFFFF', one ulp smaller than EB'(Min)'.

Section 34.4

34.4.1. The exception conditions are "o" and "u" (overflow and underflow), and the rounding mode is set to "Down" (round toward −infinity).

34.4.2. The instructions will (a) set the invalid-operation mask bit to zero, and (b) reset all the status flags to zero.

34.4.3. You would get a program interruption with IC=X'01', meaning the instruction is invalid.

Section 34.5

34.5.1. The tested data classes are:

(a) −infinity, any QNaN, and −SNaN.
(b) *Any and all* data classes, so it tells you nothing useful!
(c) Any NaN operand with any sign.
(d) A zero operand.

34.5.2. In each case, the type of the operand is inconsistent with the type of the instruction. You should analyze the operands carefully to see what each instruction "thinks" it is testing.

34.5.3. The redundant instruction is at (b): it tests for *all* data classes, so it will always set CC=1.

Section 34.6

34.6.1. The result in FPR(0,2) will be

```
FPR0: X'50123456789ABCDE'  (unchanged)
FPR2: X'FEDCBA9876543210'  (unchanged)
```

and the CC will be 2 indicating a positive result. (Compare this to the result of Exercise 33.6.1!)

34.6.2. The four results and the CC settings are:

```
(1)     LPEBR 4,2   CC=2, c(FPR4)=X'4285719600000000'
(2)     LTDBR 2,2   CC=2, c(FPR2) is unchanged
(3)     LCXBR 4,0   CC=1, c(FPR4,6)=X'8A123456789ABCDE 42857196DBB93310'
(4)     LCDBR 4,2   CC=1, c(FPR4)=X'C2857196DBB93310'
```

34.6.3. The resulting settings are (a) CC=2, (b) CC=3.

34.6.4. The TCDB instruction at (d) tests for a zero operand, so it could be replaced by

```
         LTDBR 0,0              Test c(FPR0) for zero
```

Section 34.7

34.7.2. Here's one way to calculate the table of cubes:

```
         LA    0,100            Set count of results wanted
         LE    2,=EB'1'         Constant 1 in FPR2
         LER   6,2              Initialize N
         LA    1,BCubes         Set GR1 to start of table
CubeIt   LER   4,6              Copy N to FPR4
         MEEBR 4,4              Form N**2
         MEEBR 4,6              Form N**3
         STE   4,0(,1)          Store in the table of cubes
         LA    1,L'BCubes(,1)   Increment table address
         AEBR  6,2              Increment N by 1
         JCT   0,CubeIt         Repeat for 100 values
         - - -
BCubes   DS    100EB            Table of cube values
```

34.7.3. This solution relies on setting the mask bits and the rounding mode in the FPCR:

```
        LFPC  =X'D0000003'       No underflow or inexact,
*                                round to -infinity
        LE    0,=EB'(Max)'       Maximum value
        MEEBR 0,0                c(FPR0)=X'7F7FFFFF', rounded down to (Max)
```

Section 34.8

34.8.1. As in the solution to Exercise 34.7.3, this solution relies on setting the mask bits and the rounding mode in the FPCR:

```
        LFPC  =X'D0000003'       No underflow or inexact,
*                                round to -infinity
        LE    0,=EB'1'           c(FPR0)=X'3F800000'
        DEB   0,=EB'(DMin)'      Result rounds down to (Max)
```

34.8.2. The results are:

1. X'3F800000' = -1
2. X'80000000' = -0
3. X'FF800000' = $-\infty$

34.8.3. The three values are

1. -0
2. $+0$
3. -1.0

34.8.4. Divide the signed zero into +1, and check the sign of the resulting $\pm\infty$, using a Load and Test of the appropriate operand length.

Section 34.9

34.9.1. The representation of each (DMin) is X'000...001'. When added to itself, the result is X'000...002'. The need to mask off all exception conditions is explained in the *z/Architecture Principles of Operation*; if the underflow exception is enabled, an interruption occurs and the final exponent is scaled to be in a valid range.

34.9.2. Because the value of (Min) is a power of two, adding (Min) to itself in each representation will simply double its value, by increasing the exponent by one. Thus, the results are

- X'01000000' (short precision)
- X'0020000000000000' (long precision)
- X'0002000000000000 0000000000000000' (extended precision)

34.9.3. (1) $+\infty$, (2) invalid operation.

Section 34.10

34.10.1. The CC values resulting from the comparisons are shown in this table:

Operand 1	Operand 2				
	A	B	C	D	E
A	–	2	1	2	2
B	1	–	1	2	2
C	2	2	–	2	2
D	1	1	1	–	2
E	1	1	1	1	–

Table 451. Comparing five binary floating-point operands

34.10.2. No.

Section 34.11

34.11.1. Yes, but be careful. You could "shorten" a binary floating-point operand by truncating the low-order portion, but then you can't use the truncated operand as a shorter-format operand, because the exponent field will be too long. They can be considered "truncating" only if the rounding mode is B'01' (toward zero), which chops off the low-order digits.

34.11.2. The short format has 23 significand bits; when lengthened to long format only its leading 22 or 23 bits will match those of the original long format. Thus the difference between the original and final values will be approximately $X*(2^{-22})$.

34.11.3. The long precision values will be:

1. X'3F800000 00000000' ≈ +7.8125E-3
2. X'00800000 00000000' ≈ +2.848E-306
3. X'00000001 00000000' ≈ +2.121E-314
4. X'7FE00000 00000000' ≈ +8.988E+307
5. X'7FA00000 00000000' ≈ +5.618E+306
6. X'7F800000 00000000' ≈ +1.404E+306

You must be careful to know the true length of each binary floating-point operand!

Section 34.12

34.12.2. No, because the largest 64-bit integer is less than 10^{19}, well within the range of all three binary floating-point formats.

Section 34.13

34.13.1. The results are:

1. X'40800000' = 4
2. X'40800000' = 4
3. X'40400000' = 3
4. X'40800000' = 4
5. X'40400000' = 3

34.13.2. The long precision value loaded into FPR0 is X'40230000 00000000'. But the FIEBR instruction treats it as a *short* precision operand, which has value 2.54685, so the rounded value in FPR2 is X'40400000', with value +3.

The fact that the original operand +9.5 didn't round to +10 shows why you must be careful to match operand lengths expected by the instructions to the length of the data items.

Section 34.14

34.14.1. From Table 271 on page 638 we can see that

(Max) ≈ $3.4\times10^{+38}$ so its square root is ≈ $1.84\times10^{+19}$.
(Min) ≈ 1.2×10^{-38} so its square root is ≈ 1.08×10^{-19}.
(DMin) ≈ $1.4\times10^{+45}$ so its square root is ≈ 3.74×10^{-23}.

34.14.2. Because the operand is long precision but the instruction extracts the square root of a short precision operand, the result is not +4.0, but +1.6583 instead.

Programming Problem 34.2.

This solution illustrates a way to derive useful results with low overhead.

```
P34_2    CSect ,
         Print NoGen
F0       Equ   0                 Floating-point work register
         BASR  12,0              Set base register
         USING *,12              Provide addressability
Read     ReadCard InRec,Exit     Read a record
         OC    InRec(8),=8C' '   Make upper-case characters
         MVC   PHex,InRec        Move to output line for display
         TR    Inrec(8),HexCh    Translate digits to 0-F
         PACK  DWord(5),Inrec(9) Pack into a 4-byte word
         LE    F0,DWord          Load to FPR0
         XR    1,1               Clear GR1 for decimal exponent
         MVI   ESign,C'+'        Assume positive result
         TM    DWord,X'80'       Check sign of input value
         JZ    PlusSign          Skip setting - sign, + is correct
         MVI   ESign,C'-'        Result is negative.
         NI    DWord,X'7F'       Set original sign to 0
PlusSign LPEBR F0,F0             Force argument sign + in register
         JZ    Zero              Fraction is zero, finish up easily
```

```
         CLI   DWord,X'7F'        Check possible Infinity or NaN
         JL    Finite             Skip if value is finite
         TM    DWord+1,X'80'      Check next bit
         JO    Special            If 1, it's a special value
Finite   CEB   F0,=EB'1'          See if argument has +/- exponent
         JE    FractOne           All done if exactly 1.0
         JH    PosExp             Branch if positive exponent
NegExp   MEEB  F0,=EB'1.E8'       Negative exponent; increase by 8
         AHI   1,-8               And count decimal exponent down
         CEB   F0,=EB'1'          Where did that take us?
         JL    NegExp             If it's still too small, go again
         JE    FractOne           If exactly equal, that's nice.
PosExp   CEB   F0,=EB'1.E8'       Positive exponent; check size
         JL    Reduce             If in range, reduce slowly
         MEEB  F0,=EB'1.E-8'      Multiply gives more accuracy.
         AHI   1,8                Bump decimal exponent accordingly
         J     PosExp             And do another reduction cycle
Reduce   CEB   F0,=EB'1'          Exponent in (+10,0); check for 10**0
         JE    FractOne           Fraction now is 1
         JL    Convert            Ready to convert if now a fraction
         MEEB  F0,=EB'.1'         Reduce by 10**1
         AHI   1,1                Increment exponent accordingly
         J     Reduce             And go around again
Convert  MEEB  F0,=EB'1.E5'       Convert to integer in (1,10**5)
         AEB   F0,=EB'.5'         Round it properly
         CEB   F0,=EB'1.E5'       Did it round up to 100000?
         JNL   FractOne           It did, use fraction .10000
         CFEBR 0,5,F0             Convert to binary integer in GR0
         CVD   0,DWord            Convert to packed decimal
         OI    DWord+7,X'0F'      Set correct zone
         UNPK  EDigits,DWord      Place digits into string
DoExpon  DS    0H                 Convert exponent
         MVI   ExpSign,C'+'       Assume positive exponent
         LTR   1,1                Check for correct assumption
         JNM   DoExpon2           Skip if it was right
         MVI   ExpSign,C'-'       Set exponent sign -
DoExpon2 CVD   1,DWord            Convert to decimal
         OI    DWord+7,X'0F'      Set correct (positive) zone
         UNPK  Exponent,DWord     Unpack to zoned decimal
         MVI   DecPt,C'.'         Place the decimal point
         MVI   ExponE,C'E'        And the exponent indicator
PrintIt  PrintLin PLine,Exponent-PLine+2    Print result
         J     Read               Repeat for more values
Zero     MVC   EDigits,=5C'0'     Set fraction to zeros
         J     DoExpon            And go do exponent
FractOne MVC   EDigits,=C'10000'  Set fraction digits to .100000
         AHI   1,1                Compensate by upping exponent
         J     DoExpon            And go do exponent
Special  CLC   DWord(2),=X'7F80'  Check for infinity
         JNE   ItsaNaN            Skip if not infinity
         MVC   DecPt(10),=CL10'(Infinity)'  Indicate special value
         J     Printit            Print the result
ItsaNaN  MVC   DecPt(10),=CL10'(NaN)'       Indicate special value
         J     Printit            Print the result
Exit     PrintOut *,Header=NO     Terminate
         LtOrg ,                  Insert literals
DWord    DC    D'0'               Work area
PLine    DC    C' X'''            Start of print line
PHex     DS    CL8                Input value
         DC    C''' = '
ESign    DS    C                  Sign of fraction
DecPt    DC    C'.'               Decimal point
EDigits  DS    CL5                Fraction digits
ExponE   DC    C'E'               Exponent indicator 'E'
ExpSign  DS    C                  Exponent sign
```

```
Exponent DS    CL2                Decimal exponent
HexCh    DC    256AL1(*-HexCh)    Define no-effect table
         Org   HexCh+C'A'         Set origin to offset of 'A'
         DC    X'FAFBFCFDFEFF'    Translate numeric digit to hex
         Org   ,                  Reset location counter
InRec    DS    CL80               Input area for records
         END   P34_2
```

Programming Problem 34.3. This solution displays the values in hexadecimal. You will see that many of the squared square roots match the original integer value.

```
P34_3    CSect ,
         Print NoGen
         USING *,15               Provide addressability
         LFPC  =F'0'              Mask off exceptions
         LA    0,20               Count number of values
         LE    0,=EB'1'           Constant 1 in FPR0
         LER   2,0                Integer value to be incremented
Loop     STE   2,FPInt            Store for display
         SQEBR 4,2                Calculate the square root
         STE   4,FPRoot           Store for display
         LER   6,4                Copy it to FPR6
         MEEBR 6,6                Squared square root
         SEBR  6,2                Calculate (squared root)-(integer)
         STE   6,FPDiff           Store for display
         PrintOut FPInt,FPRoot,FPDiff  Display the results
         AEBR  2,0                Increment the integer value
         JCT   0,Loop             Repeat for next value
         PrintOut *,Header=NO     Terminate
         DS    0F                 Align to fullword boundary
FPInt    DS    XL4
FPRoot   DS    XL4
FPDiff   DS    XL4
         END   P34_3
```

The only reason to mask off the exceptions is that the square roots of non-square integers are necessarily inexact.

Programming Problem 34.4. You will find the claim to be true for these values:

- Short binary floating-point: 41, 47, 55, 61, 82, 83, 94, 97
- Long binary floating-point: 49, 98
- Extended binary floating-point: 43, 86, 87, 97

Mathematically inclined readers may find it interesting to explain the discrepancies detected above, and determine why only the value 99 is shared between representations.

Programming Problem 34.5.

```
P34_5    Csect ,
         Using *,15
BFPLoop  LD    0,BFPA             Put argument in FPR0
         DDB   0,BFPX             Divide by current estimate
         SDB   0,BFPX             Subtract current estimate
         DDB   0,=DB'2'           Divide by 2
         LPDR  2,0                Copy for convergence test
         ADB   0,BFPX             Add original estimate
         STD   0,BFPX             Store updated estimate
         CDB   2,=DB'1E-10'       See if correction is small enough
         JH    BFPLoop            If not, iterate again
         ADB   0,BFPRnd           Round to 9 siginificant digits
         LDR   2,0                Copy answer to FPR2
         CFDBR 0,0,0              Convert integer part to fixed
         CVD   0,BFPDW            Convert that to packed decimal
         OI    BFPDW+7,X'0F'      Set proper zone
         UNPK  BFPAns(1),BFPDW+7(1)    Unpack the integer part
```

```
         MVI   BFPAns+1,C'.'      Place the decimal point
         CDFBR 0,0                Convert integer part back to float
         SDBR  2,0                Subtract integer part
         MDB   2,BFPCon           Convert fraction digits to integer
         CFDBR 0,0,2              Convert fraction part to fixed
         CVD   0,BFPDW            Convert that to packed decimal
         OI    BFPDW+7,X'0F'      Set proper zone
         UNPK  BFPAns+2(9),BFPDW+3(5)  Move fraction part to output
         Printout BFPAns,*,Header=NO
BFPA     DC    DB'2'              Argument
BFPX     DC    DB'1'              Initial estimate
BFPCon   DC    DB'1E9'            Conversion constant for fraction
BFPRnd   DC    DB'0.5E-9'         Constant for rounding
BFPDW    DC    D'0'               Work Area
BFPAns   DC    CL12' '            Output area for result
         End   P34_5
```

The printed result is

```
 BFPAns   = C'1.414213563 '
```

Section 35 Solutions

Section 35.0

35.0.1. I sometimes explain what I call "Cartoon" (four-finger) arithmetic by holding my hands against the edge of a table, with the thumb and first three fingers (or the four fingers) on the surface and the little fingers (or the thumbs) hidden below the edge. Then, I ask a child to count from one to "ten", where "ten" is the last finger counted: 1, 2, 3, 4, 5, 6, 7, 10. Then, I ask them to add 5 and 5.

35.0.2. Clocks use base 12 or base 24 arithmetic for hours, and base 60 for minutes and seconds; circular measure uses base 360; weekly calendars use base 7. You will enjoy browsing for others.

Section 35.1

35.1.1. Your representation could look like this:

```
 1     7                 24
┌─┬────────┬────┬────┬────┬────┬────┬────┐
│s│  char  │    6 BCD-digit fraction     │
└─┴────────┴────┴────┴────┴────┴────┴────┘
```

Because the characteristic is 7 bits wide, it can accommodate characteristic values from 0 to 127. To avoid the range asymmetry of hexadecimal floating-point data, EMax should be +64 and EMin should be −63. Then, the characteristic bias is +63. (See the footnote on page 681.)

35.1.2. The three quanta are 1×10^{7}, 1×10^{6}, and 1×10^{3} respectively.

35.1.3. Yes. The quantum in each case is 10^{exponent}.

35.1.4. The number of significant digits is 5, 6, and 9 respectively.

35.1.5. The pictured representation has precision of 19 digits, and the given value has five nonzero digits between its leftmost and rightmost nonzero digits, so the cohort has 15 members in each case.

35.1.6. $p-n+1$.

Section 35.2

35.2.1. The cohort members and their quanta are

1. [−2‖0743] and [−3‖7430], with quanta 1×10^{-2} and 1×10^{-3} respectively.
2. [+1‖0009], [0‖0090], [−1‖0900], and [−2‖9000], with quanta 1×10^{1}, 1×10^{0}, 1×10^{-1}, and 1×10^{-2} respectively.
3. [−3‖3400], [−2‖0340], and [−1‖0034], with quanta 1×10^{-3}, 1×10^{-2}, and 1×10^{-1} respectively.

35.2.2. 2, 5, and 11. (That wasn't too hard, was it!)

35.2.3. Nine. A long-precision number supports 16 digits, so we can have from 0 to 8 high-order (or low-order) zeros.

35.2.4. Assuming all exponents are valid, you can't tell unless you know how many low-order zero digits are in the significand. For example, if the significand is 10000000, its cohort has 16 members. Similarly, if the significand is 12345678, its cohort has 9 members.

35.2.5. The encoding is `X'38F'`.

35.2.6. The BCD digits are 945.

35.2.7. The values are:

1. `X'827C000000000000'` = −0: first 5 CF bits = 00000, TSF=0 bits = 11111 1
2. `X'7EF00049826BA3B0'` = +SNaN: first 5 CF bits = 11111, next bit = 1 bits = 11111 0
3. `X'FB7388215142D357'` = −QNaN: first 5 CF bits = 11111, next bit = 0

35.2.9. The two values are

```
Short DFP:  X'2DF306BB'
Long  DFP:  X'22200000003306BB'
```

35.2.10. Examine Table 320 on page 685: for decimal digits 0-7, the bit patterns in the rightmost encoded hexadecimal digit are the same as those of the “source” digit. Decimal digits 8 and 9 may sometimes be encoded to X'E' and X'F' respectively.

Section 35.3

35.3.1. Seven possible interpretations, and their Assembler Language defining statements, are:

```
         DC    F'1077952604'      Word binary integers
         DC    C'•••*'            Characters (three blanks and an asterisk)
         STH   4,X'05C'(0,4)      Store Halfword
         DC    P'+4040405'        Packed decimal
         DC    E'.250982046'      Hexadecimal floating-point
         DC    EB'3.0039282'      Binary floating-point
         DC    ED'1.0850E+35'     Decimal floating-point
```

The decimal floating-point value is a member of a cohort: members with the same value can have different representations:

```
         DC    ED'10850E31'       Representation X'4040405C'
         DC    ED'1085E32'        Representation X'4050044B'
         DC    ED'1.085E35'       Representation X'4050044B'
```

The last two of these show that the presence of the low-order zero digit in the nominal value changes the generated constant.

35.3.2. The three constants are

```
Short    DC    ED'.9999998E97'    Generates X'77F3FCFE'
Long     DC    DD'.99999999999998E385'
Extend   DC    LD'.999999999999999999999999999999998E6145'
```

It's instructive to assemble these three constants together with the following three (specify the PRINT DATA statement!). You'll see that the results for the first three constants are identical to the Max values except that the low-order bit is one less.

```
         DC    ED'(Max)'          Generates X'77F3FCFF'
         DC    DD'(Max)'
         DC    LD'(Max)'
```

35.3.3. The three constants are

```
Short    DC    ED'.9999999E97'
Long     DC    DD'.9999999999999999E385'
Extend   DC    LD'.9999999999999999999999999999999999E6145'
```

If you don't like writing it as a fraction, you could also write (for example)

```
Long     DC    DD'9999999999999999E369'
```

35.3.4. The sketches of the hexadecimal and binary floating-point representations in Figure 437 on page 681 imply that those representations can safely be lengthened or shortened. However, decimal floating-point values depend on maintaining the precise structure of the Combination Field and of the declets in the Trailing Significand Field. Truncating or padding a constant could produce a bit string unrecognizable as a decimal floating-point value.

35.3.5. Because short decimal floating-point exponents range from −95 to +95, the cohort of +0 has 192 members, not 191: remember that +0 and −0 are *different* members of the cohort.

35.3.7. NMin = 1.E-95, and DMin = .000001E-95.

Section 35.4

35.4.1. The tested data classes are:

(1) −infinity, any QNaN, and −SNaN (The Effective Address bit pattern is B'11101')
(2) *Any and all* data classes, so it tells you nothing useful!
(3) Any NaN operand with any sign.
(4) A zero operand.

Section 35.5

35.5.1. Yes: if the result has fewer significant digits than the representation provides, the cohort member with smallest quantum will have a nonzero leftmost digit.

35.5.2. A program interruption with IC=1 (operation exception) will occur.

Section 35.6

35.6.1. Both instruction sequences do the same, and both sequences require four bytes. Depending on the System z model, the single instruction could be slightly faster than the two LDRs.

Section 35.7

35.7.1. The results are

```
(1) X'0000000000000000' = +0
(2) X'7800000000000000' = +infinity
(3) X'7800000000000000' = +infinity
(4) X'F800000000000000' = -infinity
```

Section 35.8

35.8.1. The CC settings are shown in this table:

Operand 1	Operand 2		
	Finite	**Infinity**	**NaN**
Finite	1 or 2	3	3
Infinity	3	0	3
NaN	3	3	0

Section 35.9

35.9.1. To avoid an invalid operation exception, the rounding masks must round the decimal floating-point value toward zero.

- For Mp, the mask values are 9, 11, 13, and 15 (and 0 if the Decimal Rounding Mask in the FPCR has values 1, 3, 5, or 7).
- For Mn, the mask values are 8, 9, 10, 13, and 15 (and 0 if the Decimal Rounding Mask in the FPCR has values 1, 2, 5, or 7).

There is one more mask value for Mn because -2^{63} is even, while $+2^{63}-1$ is odd, and can round (up) to an even final digit, making the result larger than the maximum positive integer.

Section 35.10

35.10.1. You could compare the decimal floating-point operand to a constant. Suppose a nonnegative long operand is in FPR2:

```
         LD    0,=DD'1E16'        10**16 (16 significant digits)
         CDTR  2,0                Compare operand to constant
         JNL   LostDigit          Branch if a digit will be lost
```

and similarly for extended precision, where the constant would be =DD'1E32'.

Alternatively, you could use the “Extract Significance” instructions to determine the actual number of significant digits.

35.10.2. Because the result is unsigned, there is no need to choose a sign code.

35.10.3. Because the source operand in (GG8,GG9) has at most 31 packed decimal digits, but the extended decimal floating-point operand supports 34 digits, the LMD must necessarily be zero.

35.10.5. Let ZD represent the numeric value of the zoned decimal digits of the result.

S	Z	P	F	Resulting behavior
0	0	—	—	All zones are X'F'
0	1	—	—	All zones are X'3'
1	0	0	0	All zones are X'F', sign code is F or D
1	0	1	0	All zones are X'F', sign code is C or D
1	—	0	1	Zones are F'3' or X'F', and if |ZD|=0, sign code is F
1	—	1	1	Zones are F'3' or X'F', and if |ZD|=0, sign code is C
1	1	0	0	Zones are F'3' and sign code is C
1	1	1	0	Zones are F'3' and sign code is F

Section 35.11

35.11.2. The value of 24÷7 is $3.\underline{428571}$, where the underlined group of digits is repeated indefinitely.

35.11.3. Use these two instructions to load the complement of the extended operand in (FPR0,FPR2) to (FPR4,FPR6):

```
LCDR  4,0             Complement high-order half
LDR   6,2             Copy low-order half
```

35.11.4. If the operand is a SNaN, Load and Test will cause an invalid operation exception, but Test Data Class will not.

Section 35.12

35.12.1. The results are:

```
(1)     SLDT  2,2,0     Significance = 8
(2)     SLDT  2,2,3     Significance = 11
(3)     SRDT  2,2,6     Significance = 2
(4)     SLDT  2,2,10    Significance = 16
(5)     SRDT  2,2,8     Significance = 0
```

35.12.2. Remember that the significand of a DFP number is encoded in declets. Thus, the low-order 24 bits of the DFP number are X'2500000', representing a declet X'025' followed by two X'000' declets. Referring to Figure 470 on page 735, their values are 025, 000, and 000, so there are 8 significant digits.

35.12.3. An advantage of using shifts might include simplicity. Some disadvantages include (a) lack of rounding choices on right shifts and (b) lack of overflow detection on left shifts.

35.12.4. Consider

```
LD    0,=DD'12789'       Source operand = X'2238000000004BCF'
LD    2,=DD'12789E+07'   Second operand = X'2254000000004BCF'
QADTR 4,0,2,8                  c(FPR4) = X'2254000000000000'
```

The result has zero significand and nonzero exponent.

35.12.5. The Assembler limits the length of P-type constants to 16 bytes.

Section 35.14

35.14.1. Consider the value X'77FFFFFF' Following the zero sign bit, the first two 'ab' bits are 11, so the biased exponent is formed from all but the last CF bit (cdefghij), giving B'10111111' = X'BF'=191. Subtracting the exponent bias (101) means that the exponent is 90. The significand's high-order bits are B'1001' (8+k) so the significand is X'9FFFFF'=10485760. But the largest valid significand is 10^7-1 = X'96967F'=9999999, so the significand is too large. The value is non-canonical, and will be treated as zero.

35.14.2. Using the values calculated in Exercise 35.14.1, we can easily construct the Max value: X'7796967F'.

35.14.3. X'00000001'. The biased exponent is zero, so the true exponent is −101, and the significand is 1. Thus its value is 1×10^{-101}.

35.14.4. Starting with X'5FFFFFFF', the 'ab' bits are 10, so the biased exponent is formed from bits abcdefgh, giving B'10111111', or X'BF'=191 for the biased exponent. Subtracting the bias (101) means the exponent is 90. The first 3 bits (ijk) of the significand are 111, so the significand is X'7FFFFF' = $2^{23}-1$=8388607, which is less than 9999999, so this is a valid representation of 8388607×10^{90}.

Section 35.15

35.15.1. The M_4 mask is used in four different ways:

M_4	Use and Meaning
1	Used by CSDTR, CSXTR to select the sign code of the packed decimal result.
4	For FIDTR and FIXTR; and for FIEBRA, FIDBRA, and FIXBRA; and for CEFBRA, CDFBRA, CXFBRA, CEGBRA, CDGBRA, and CXGBRA; and for CFEBRA, CFDBRA, CFXBRA, CGEBRA, CGDBRA, and CGXBRA; and for CELFBR, CDLFBR, CXLFBR, CELGBR, CDGBRA, and CXLGBR; and for CDLGTR, CXLGTR, CDLFTR, and CXLFTR; and for CGDTRA, CGXTRA, CFDTR, and CXFTR; suppresses the inexact exception.
8	For LDETR and LXDTR, to indicate an invalid operation exception; for LEDTR and LDXTR, to truncate the payload.
any	Used by Add (ADTRA, AXTRA), Divide (DDTRA, DXTRA), Divide to Integer (DIEBR, DIDBR), Multiply (MDTRA, MXTRA), Quantize (QADTR, QAXTR), Reround (RRDTR, RRXTR), and Subtract SDTRA, SXTRA), instructions as a 4-bit rounding mask.

Programming Problem 35.2. The values of N that fail for both long and extended precision decimal floating-point arithmetic are 3, 9, 22, 28, 33, 34, 45, 48, 49, 55, 63, 65, 66, 74, 75, 84, 88, 90, 95, and 99.

The values of N that fail only for long precision decimal floating-point arithmetic are 31, 41, 43, 51, 59, 71, 73, 82, 85, 92, 93, and 98.

The values of N that fail only for extended precision decimal floating-point arithmetic are 47, 67, 83, 86, and 89.

The failing values share the property that they are divisible by a prime number other than 2 or 5.

Programming Problem 35.3. This solution uses a specialized interface between DPD2BCD and its caller: the input and returned values are in the left and right halves of a word argument.

```
         Title 'DPD2BCD: Convert a 10-bit declet to 3 BCD digits'
*  Entry via R15, return via R14; R1=A(A(fullword)), where the
*  first 2 bytes of the fullword are the (right-justified)
*  declet, and the last 2 bytes are the 3 converted BCD digits,
*  also right-justified.
*  Declet format:    PQ RSTU VWXY
*  BCD digits:     ABCD EFGH IJKM

DPD2BCD  CSect ,
DPD2BCD  RMode Any
DPD2BCD  AMode 31
         Using *,15
         STM   14,4,12(13)        Save some regs
         L     3,0(0,1)           A(fullword)
         Using Fullword,3
         LH    0,Declet           Get the declet
         LR    4,0                Carry declet in R4
         XR    1,1                Clear R1
         SRDL  0,1                Isolate Y bit (=M)
         SRL   0,3                Drop VWX bits
         SRDL  0,1                Isolate U bit (=H)
         SRL   0,2                Drop ST bits
         SRDL  0,1                Isolate R bit (=D)
         SRL   1,28               Now have 2*DHM bits
         LH    2,DHMBCD(1)        Carry BCD digits in R2
*                                 RUY=DHM bits now completed
         SLL   0,9                PQ bits positioned 0PQ0 0000 0000
*                                 0PQ(R) bits for BCD1: ABC(D)
         TML   4,VBit             V = 0? (VWXST bits 0----)
         JZ    Case_A             Jump if yes
*                                 Know V=1
         LA    1,B'1100000'       Prepare for cases B and D
         TML   4,WBit+XBit        WX=00? (100--)
```

```
         JZ    Case_B            Jump if yes
*                                Know V=1,XW not 00
         TML   4,WBit            W=0? (10---)
         JZ    Case_C            Jump if yes
*                                Know V=1, W=0
         TML   4,XBit            X=0? (110--)
         JZ    Case_D            Jump if yes
*                                Know V=1, X=1, W=1
         TML   4,SBit+Tbit       ST=00 or 11?
         JZ    Case_E            Jump if ST=00 (11100)
         JO    Case_H            Jump if ST=11 (11111)
*                                Know VXW=111, and ST mixed
         TML   4,SBit            S=0?
         JZ    Case_F            Know XVW=11, S=0, T=1
Case_G   DC    0H                Do OPQ(R), 100(U), 100(Y)
         OR    2,0               Complete the OPQ(R) bits
         STH   2,BCD
         OI    BCD+1,X'80'+X'08' Do 100U = 100H, 100Y = 100M
         J     Finis             Return
Case_A   DC    0H                Do OPQ(R), OST(U), OWX(Y)
         OR    2,0               Complete the OPQ(R) bits
         LA    1,B'1100110'      Extract ST and WX bits
         NR    4,1               Isolate ST and WX bits
         OR    2,4               Insert the ST and WX digits
         STH   2,BCD
         J     Finis             Return
Case_B   DC    0H                Do OPQ(R), OST(U), 100(Y)
         OR    2,0               Complete the OPQ(R) bits
         NR    4,1               Extract ST bits
         OR    2,4               Insert
         STH   2,BCD
         OI    BCD+1,X'08'       Do 100Y = 100M
         J     Finis             Return
Case_C   DC    0H                Do OPQ(R), 100(U), OST(Y)
         OR    2,0               Complete the OPQ(R) bits
*                                Do OST(Y) inline
         SRL   4,4               Position for 3rd digit
         LA    1,B'110'          Extraction bits for ST
         NR    4,1
         OR    2,4               Insert the ST digits
         STH   2,BCD
         OI    BCD+1,X'80'       Do 100U = 100H
         J     Finis             Return
Case_D   DC    0H                Do 100(R), OST(U), OPQ(Y)
         SRL   0,8               Reposition PQ bits
         OR    2,0               Done with OPQ(Y)
         NR    4,1               Extract ST bits
         OR    2,4               Insert
         STH   2,BCD
         OI    BCD,X'08'         Do 100R = 100D
         J     Finis             Return
Case_E   DC    0H                Do 100(R), 100(U), OPQ(Y)
         SRL   0,8               Reposition PQ bits
         OR    2,0               Insert the bits
         STH   2,BCD
         OI    BCD+1,X'80'       Do 100U = 100H
         OI    BCD,X'08'         Do 100R = 100D
         J     Finis             Return
Case_F   DC    0H                Do 100(R), OPQ(U), 100(Y)
*                                Do OPQ(U) inline
         SRL   0,4               Position PQ bits for 2nd digit
         OR    2,0               Insert the PQ digits
         STH   2,BCD
         J     Case_H_1          Complete the rest
Case_H   DC    0H                Do 100(R), 100(U), 100(Y)
         STH   2,BCD             Store DHM bits
```

```
         OI    BCD+1,X'80'        Do 100U = 100H
Case_H_1 DC    0H
         OI    BCD,X'08'          Do 100R = 100D
         OI    BCD+1,X'08'        Do 100Y = 100M
         J     Finis              Return
Finis    DC    0H
         LM    14,4,12(13)
         BR    14
         Drop  15
DHMBCD   DC    XL2'000,001,010,011,100,101,110,111'

Fullword DSect ,
Declet   DS    0H
         DS    X                  0000 00PQ
         DS    X                  RSTU VWXY
SBit     Equ   X'40'
TBit     Equ   X'20'
VBit     Equ   X'08'
WBit     Equ   X'04'
XBit     Equ   X'02'
BCD      DS    H                  Digits: 0, BCD1, BCD2, BCD3
         End
```

A sample program to print the table of all possible declet values and their converted results:

```
P35_3    CSect ,
P35_3    RMode 24                 Only needed to use Printlin and...
P35_3    AMode 24                 Printout macros.
         Using *,15
         STM   14,12,12(13)       Save registers
         LA    12,Savearea        Point to save area
         ST    13,4(,12)          Chain caller's area to ours
         ST    12,8(,13)          Chain ours to caller's
         LR    13,12              R13 now points to ours
         LR    12,15              Set new base address
         Drop  15
         Using D2B,12
NTry     Equ   1023               Do declet values from 0 to 1023
         LHI   10,NTry            Set loop counter
         LHI   9,8                Number of values per line
         LA    8,Line             Point to the print line
         MVC   Line,Line-1        Propagate blanks
Loop     DC    0H
         LHI   3,NTry             Load maximum number to convert
         SR    3,10               Subtract current loop count
         STH   3,Input            Store declet value in argument
         LA    1,ArgAddr          Point to it
         L     15,SubAddr         Get address of DPD2BCD
         BASR  14,15              Call the subroutine
*
         CVD   3,DW               Convert declet value to packed
         Using Out,8              Map output group onto print line
         UNPK  N,DW+5(3)          Unpack the decimal value
         OI    N+L'N-1,X'F0'      Set correct zone on last digit
         CLI   N,C'0'             Blank leading zeros
         JNE   DoD                Jump if none
         MVI   N,C' '             Blank leading zero
         CLI   N+1,C'0'           Check next digit
         JNE   DoD                Jump if not zero
         MVI   N+1,C' '           Blank it
         CLI   N+2,C'0'           Check next digit
         JNE   DoD                Jump if not zero
         MVI   N+2,C' '           Blank it otherwise
DoD      UNPK  DU,Input(3)        Unpack the declet's three digits
         TR    D,HEX-C'0'         Translate to hex
         MVI   D+L'D,C' '         Blank the trailing 'swap' byte
```

```
         UNPK  BU,Output(3)        Unpack the declet's decimal value
         TR    B,HEX-C'0'          Translate to hex
         MVI   B+L'B,C' '          Blank the trailing 'swap' byte
         LA    8,OutL(,8)          Step to next output-line field
         JCT   9,Next              Loop back for more if line not full
         Printlin Line             Output the print line
         MVC   Line,Line-1         Clear the print line
         LHI   9,8                 Reset the number of values per line
         LA    8,Line              Point to start of print line
*
Next     AHI   10,-1               Count number of conversions
         JNM   Loop                If not negative yet, repeat
         CHI   9,8                 Did the last line yet?
         JE    Done                If yes, we're done.
         Printlin Line             Otherwise, output the last line
Done     Printout *,Header=NO      Terminate
SubAddr  DC    V(DPD2BCD)
ArgAddr  DC    A(Input)            Address of the argument to DPD2BCD
Input    DS    H                   Input value in left halfword
Output   DS    XL2                 Output value in right halfword
Savearea DS    18F
DW       DS    D                   Work area for CVD
         DC    C' '
Line     DC    CL121' '
HEX      DC    C'0123456789ABCDEF'
Out      DSect ,                   Mapping of an output group
         DS    C
N        DS    CL4,C
DU       DS    0CL4
D        DS    CL3,C
BU       DS    0CL4
B        DS    CL3,C
OutL     Equ   *-Out
         End   P35_3
```

The last line of printed output looks like this:

```
1016 3F8 778  1017 3F9 779  1018 3FA 796  1019 3FB 797  1020 3FC 976  1021 3FD 977  1022 3FE 998  1023 3FF 999
```

Section 36 Solutions

Section 36.1

36.1.1. When converting short-precision data from hex to binary, some magnitudes may be unrepresentable; when converting from binary to hex, from 1 to 3 bits of precision may be lost, and special values cannot be converted.

36.1.2. When converting long-precision data from hex to binary, 1 to 3 bits of precision may be lost; when converting from binary to hex, some magnitudes may be unrepresentable, and special values cannot be converted.

36.1.3. When converting extended-precision data from hex to binary there are no problems, but when converting from binary to hex there may be unrepresentable magnitudes, loss of 1 to 4 bits of precision, and unrepresentable special values.

Section 36.2

36.2.1. The result is 3.00001 in both cases.

36.2.2. The result is 3.00000 in both cases.

Section 36.4

36.4.1. A usable hexadecimal floating-point value requires at least one hex digit in the fraction. Consider two other constants:

```
E1       DC    EL.9'1'
E15      DC    EL.9'15'
```

The nominal value of each is one hexadecimal digit; but the only significant bit (B'0001') in the fraction of the constant named **E1** is lost, so the assembler will flag this statement. Similarly, the constant named **E15** is rounded up to appear to have nominal value 16, which then has the same fraction digit as the first constant.

36.4.2. The binary constant interpreted as a hexadecimal constant would appear to have value +1.953125E−4, and the decimal constant would appear to have value +1.880791E−37! This shows why you must be careful not to mix floating-point data types.

36.4.3. The BFP value is +2.4, and the DFP value is +4613008E+20.

36.4.4. The HFP value is +2.2350989E−37, and the BFP value is +2.818926E−18. (Neither seems very close to 1.)

36.4.5. The values of the four constants are:

Bit Pattern	**HFP Value**	**BFP Value**	**DFP Value**
(1) X'3F800000'	0.03125	1.0	7.000000E25
(2) X'41100000'	1.0	9.0	0
(3) X'42640000'	100.0	57.0	2.000000E70
(4) X'7FFFFFFF'	MaxReal	QNaN	SNaN

Be *very* careful when mixing data types; remember that most bit patterns are valid in FP instructions.

Section 36.5

36.5.1. All the constants are represented by X'40100001'.

36.5.2. The result is 0.625000596...E−1.

Section 36.6

36.6.1. Any finite unnormalized value A such as X'00010000' with characteristic zero will fail. If exponent underflow is masked off the result is X'00000000', and if exponent underflow occurs the result is X'7F100000'. Neither result equals the original value of A.

36.6.2. If A is a QNaN or SNaN, the result will be “Not any Number”. (You might want to claim that the result is the value you started with; but remember that the “law” stated in Table 396 applies only to numbers!)

36.6.3. Several unnormalized values of A show this behavior:

A	A×(1/A)	ulp difference
X'4100000F'	X'40FFFFFF'	16
X'4100000C'	X'40FFFFFC'	16
X'4100000A'	X'40FFFFFA'	16

36.6.4. HexMax = 16^{+63}−ulp, and HexMin = 16^{-65}, so their product is approximately 16^{-2}, or 1/64.

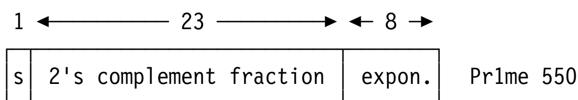
Section 36.7

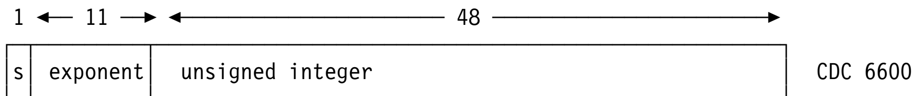
36.7.1. There are 127 nonzero characteristic values and two sign values, so you can create 254 pseudo-zeros in each representation.

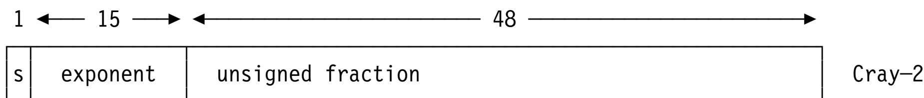
36.7.2. Because the characteristics differ by 1, the first operand fraction will be shifted right by one digit position. That digit will be shifted left from the guard digit position, so the result will be the same as the first operand.

Section 36.8

36.8.1. The formats would look like this:







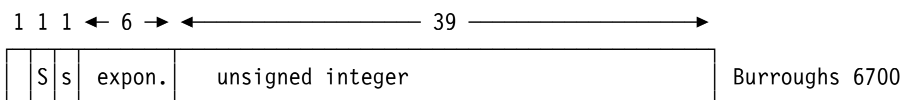
36.8.2. Consider adding a tiny unnormalized value to zero:

```
         LZER  0                   Set FPR0 to zero
         AE    0,=X'00000003'      Add a tiny unnormalized value
```

If program interruptions are enabled the result is an exponent underflow, and the result in FPR0 is X'7B300000' (not exactly equal to X'00000003'!). If disabled, the result is zero.

36.8.3. Consider z=X'44200000' and y=X'47000200'. These two values compare equal, so z = y. Now, if If t=X'42000010', we find that z−t=X'441FFFFF' and y−t=X'47000200', so $z-t \neq y-t$.

36.8.4. The Burroughs 6700 used a 48-bit word;* its short floating-point representation had one unused high-order bit followed by the significand sign bit S and the exponent sign bit s:



36.8.5. The values are shown in this table:

Hex	Char	Integer	BFP	HFP	DFP
X'81818181'	C'aaaa'	-2122219135	-4.7572945E-38	-6.9902216E-77	-6.0301E-73
X'A3A3A3A3'	C'tttt'	-1549556829	-1.774180E-17	-7.694278E-36	-1.68723E+26
X'F5F5F5F5'	C'5555'	-168430091	-6.235846E+32	-6.323899E+63	-9.873375E+64
X'FEFEFEFE'	C'ÚÚÚÚ'	-16843010	-1.694740E+38	-4.505390E+74	-SNan

* Words were actually 52 bits long: the four high-order bits were a parity bit and three reserved "tag" bits. Application programmers saw only the 48 bits.

Programming Problem 36.3.

Long BFP values have greater exponent range than long HFP values, and also support special values; these cases must be detected. Because long BFP operands are 53 bits long while HFP operands are 56 bits long, no rounding is required.

Section 37 Solutions

Section 37.1

37.1.1.

```
ShftRt   SRL   0,2               Preliminary shift by 2
         L     1,NN              Shift amount in GR1
         LTR   1,1               Check sign of n
         BNPR  14                Return immediately if not +
         SRL   0,0(1)            Perform remaining shifts
         BR    14                And return
```

37.1.2.

```
         L     0,Logic           Get shift-able argument
         L     1,NN              And shift amount
         SRL   0,2               Do fixed part of shift
         LTR   1,1               Test variable part
         BNP   Store             Skip shift if not +
         SRL   0,0(1)            Shift by variable amount
Store    ST    0,Result          Store the answer
         BR    14                And return to caller
```

37.1.3. You will remember from Section 24.11 that when a Branch and Save instruction is decoded, the IA still contains the address of the instruction following the EX. Thus, the IA from the PSW placed in R_1 will contain *that* address, *not* the address of the instruction following the Branch and Save. The next instruction fetched will be at the branch address, unless the executed instruction was `BASR r,0`.

37.1.4. You probably found the error immediately: "`BASR 2,0`" doesn't branch, but "`BASR 2,2`" branches to whatever address was in GR2 before the instruction was executed. The error is caused by not understanding the selection of the branch address *before* the Instruction Address is placed into GR2.

37.1.5. This is a cute way to execute blocks of code exactly twice, without looping. The first "`BASR 3,0`" sets GR3 to the address of the following instruction, and then the first block of code is executed. When the "`BASR 3,3`" is executed, control returns to the start of the first block of code, but GR3 now contains the address of the first instruction of the *second* block of code. When control again reaches the first "`BASR 3,3`", the branch address in GR3 is the address of the next instruction, so the "`BASR 3,3`" now acts like it was a "`BASR 3,0`". The second block of code is executed twice, in the same way.

37.1.6. If the Location Counter was on an odd halfword boundary when the BAS instruction is assembled, two empty bytes will be skipped by the Assembler so that the address constants will be properly aligned on a word boundary. A branch to `8(,14)` will then arrive in the middle of the second address constant.

37.1.7. This exercise illustrates an extreme antisocial tendency on someone's part. The calling program can re-establish its own registers only if (1) it had saved all of them within an addressable distance from the return point, and (2) the instruction at the return address is a BASR x,0 that establishes a temporary base register that can be used to reload the registers. For example:

```
         BASR  14,0              Local temporary base register
         Using *,14              Establish addressability
         LAY   14,MySave         Hope our save area is addressable!
         Drop  14                No local base register now
         LM    0,15,0(14)        Restore the registers
```

Now you see why the callee should save and restore registers!

37.1.8. instructions BASR R_1,R_2 and BAS R_1,0(0,R_2) are identical only if R_2 is not zero; the BAS instruction would then branch to memory location zero. Otherwise, they are identical.

When we consider that BCR R_1,0 and BCTR R_1,0 behave similarly in not branching, we might say that the RR forms of the three instructions aren't needed unless we need an instruction to place the IA in a register, or a two-byte no-operation, or an instruction to reduce a register's contents by one. At the cost of destroying some of the symmetry of the System/360 instruction set, BASR, BCTR, and BCR could be removed and replaced by a single RR-type instruction in which the R_2 digit specified which of the three non-branching activities was intended.

37.1.9. The third instruction will be executed indefinitely. The first BASR instruction puts the address of the second BASR in GR5, and then the second BASR will branch to its own address. GR4 will contain the address of whatever follows the second BASR.

37.1.10. Consider these instructions:

```
ShftRt4A STM   1,2,ShftJG1         Save GR1 through GR2
         LM    1,2,0(1)            Get argument addresses
         L     0,0(0,1)            Get first argument in GR0
         L     1,0(0,2)            Get second argument in GR1
         LTR   1,1                 Test sign of shift amount
         JNM   ShftOK              Branch if non-negative
         SR    1,1                 Set shift amount to zero
ShftOK   SRL   0,2(1)              Shift by required amount
         LM    1,2,ShftJG1         Restore GR1-GR2
         BR    14                  Return to caller
ShftJG1  DS    2F                  Save area for 2 registers
```

37.1.11.

```
ShftRt4B ST    1,ShftJG2           Save GR1
         L     1,0(0,1)            Get first argument address
         L     0,0(0,1)            Get first argument in GR0
         L     1,ShftJG2           Restore GR1
         L     1,4(0,1)            Get second argument address
         L     1,0(0,1)            Get second argument in GR1
         LTR   1,1                 Test sign of shift amount
         JNM   ShftOK              Branch if nonnegative
         SR    1,1                 Set shift amount to zero
ShftOK   SRL   0,2(1)              Shift by required amount
         L     1,ShftJG2           Restore GR1
         BR    14                  Return to caller
ShftJG2  DS    F                   Save area for 1 register
```

This solution is probably less efficient than that in Exercise 37.1.10 because (a) it has more instructions, and (b) it must load GR1 from the save area twice.

Section 37.3

37.3.1.

```
ShftRt   STM   1,3,SaveRegs        Save working registers
         LM    1,3,0(1)            Get all argument addresses
         L     1,0(,1)             Get word to be shifted
         L     2,0(,2)             Get shift count N
         LTR   2,2                 Check sign of shift count
         JNM   ShftOK              Skip next instruction if nonnegative
         SR    2,2                 Set count to zero if negative
ShftOK   SRL   1,2(2)              Shift 2+max(N,0) places
         ST    1,0(,3)             Store result at specified place
         LM    1,3,SaveRegs        Restore caller's registers
         BR    14                  And return
SaveRegs DS    3F                  Save area for registers GR1-GR3
```

37.3.2. If more than one element has the same maximum value, which will appear in GR0 on return?

```
AMax     STM   1,2,Saver           Save GR1, GR2
         LM    1,2,0(1)            Get array and length addresses
         L     0,0(,1)             Load 1st element of array
         L     2,0(,2)             Load length of array
         CHI   2,1                 See if array is short (1 element)
         JNH   Exit                Return if 1 (or fewer) items
Comp     C     0,0(,1)             Compare current max to an element
         JNL   Skip                Skip if max is still a max
         L     0,0(,1)             Else element is the new max
Skip     LA    1,4(,1)             Step to next element of array
         JCT   2,Comp              And try again
Exit     LM    1,2,Saver           Restore registers
         BR    14                  And return
Saver    DS    2F                  Save area for GR1, GR2
```

37.3.4. This solution counts the bits using logical addition.

```
NBits    CSect ,                 Count 1-bits in a byte string
         Using *,15              Local base register
         STM   0,4,LocalSav      Save GR0-GR4 locally
         LM    1,3,0(1)          Get argument addresses
         XR    4,4               Initialize bit count
         L     2,0(,2)           Get string length
         LTR   2,2               Check length
         JNP   Store             If not positive, return zero
         LR    0,4               Clear GR0 for byte insertion
Loop     ICM   0,B'1000',0(1)    Get a byte from the string
Test     ALR   0,0               Add GR0 to itself, check for carry
         JZ    Next              If c(GR0)=0, no 1-bits in the byte
         JM    Test              CC=1 means no carry, bit was zero
         LA    4,1(,4)           Increment the count
         JO    Test              Repeat for next 1-bit
Next     LA    1,1(,1)           Step to next byte
         JCT   2,Loop            Count down, get another byte if any
Store    ST    4,0(,3)           Store bit count
         LM    0,4,LocalSav      Restore registers
         BR    14                Return to caller
LocalSav DS    5F                Save area for GR0-GR4
         End
```

37.3.5. Suppose you wrote

```
         ORG   *,8,-2
```

If the LC is already aligned on a doubleword boundary, it would remain at the same location; the "offset" operand -2 would then subtract 2 from the LC, meaning that the Y-con might overlay the last 2 bytes of whatever preceded the doubleword boundary.

37.3.8. Assuming this is the calling sequence:

```
         LA    1,ArgList         Point to argument addresses
         BRAS  14,ShftRt64       Call the subroutine
         - - -
ArgList  DC    AD(W,N)           A(data, shift amount)
         DC    FD'-1'            'Fence' of all 1-bits
W        DC    X'ABCDEF95'       Data to be shifted
N        DC    F'3'              Shift amount
```

then this is a possible solution:

```
ShftRt64 STMG  1,2,ShftSave      Save GG1, GG2
         LMG   1,2,0(1)          Get argument addresses
         L     0,0(,1)           Load data item in GR0
         LT    1,0(,1)           Load shift amount in GR1
         JNM   ShftOK            If nonnegative, go to shift
         SR    1,1               If -, set shift amount to zero
ShftOK   SRL   0,2(1)            Shift by required amount
         LMG   1,2,ShftSave      Restore caller's registers
         BR    14                Return to caller
ShftSave DS    2D                Save area for GG1, GG2
```

Section 37.4

37.4.1. The low-order bit of the character A is 1; but the back chain address of a standard save area points to a word address, so its two low-order bits would be zeros.

37.4.2. If the caller had entered your routine in 64-bit addressing mode, he would have provided a Format-4 save area. Because your routine was called in 24- or 31-bit addressing mode, the return address doesn't depend on the high half of GG14.

37.4.3. The lengths are X'48', X'90', and X'D0' bytes respectively.

Section 37.5

37.5.1. In the early days, only BALR and BAL were available; BASR and BAS were introduced later. The BALR and BAL instructions put the right half of the PSW into the R_1 register. In 24-bit addressing mode, the leftmost byte of the register always contains the Instruction Length Code (2 bits), the Condition Code (2 bits) and the Program Mask (and may therefore be slower than BASR and BAS). The rightmost 6 bits of that byte could be all 1-bits, but because BALR and BAL are respectively 2-byte and 4-byte instructions, the ILC will never be B'11'.

When 31-bit addressing mode was introduced, executing

```
         MVI   12(13),X'FF'
```

would have destroyed the high-order 7 bits of the return address in the save area (after GR14 had been loaded from the save area!), making it difficult to know where the call came from. This was solved by setting the low-order bit of the return address to 1.

Be careful, though: you should set a return flag only if the calling and called routines both execute in 24- or 31-bit addressing mode.

37.5.2. It does conform to standard linkage conventions: GR13 contains the address of a save area; GR14 contains the return address; GR15 contains the entry point address of the subroutine; and GR1 contains the address of the argument list. Also, the calling-point identifier is at the return address.

This isn't standard *coding* practice, though.

37.5.3. The instructions could be revised like this:

```
         BASR  14,15              Link to subroutine as usual
         LTR   15,15              Test sign of return code
         JM    Set12              If negative, set to 12
         CHI   15,12              Compare to 12
         JH    Set12              If greater, set to 12
         TMLL  15,X'3'            Test two low-order bits of retcode
         JZ    Proceed            If 0, multiple of 4; OK to proceed
         AHI   15,X'3'            Force carry to round to next value
         NILF  15,X'FFFFFFFC'     Set 2 low-order bits of GR15 to zero
         J     Proceed            OK to branch to return code handler
Set12    LA    15,12              Return code forced to 12
Proceed  LARL  14,JList           Address of branch list
         AR    14,15              Add return code to list address
         BR    14                 One-hop branch to correct routine
JList    J     Ret000             Return code = 0
         J     Ret004             Return code = 4
         J     Ret008             Return code = 8
         J     Ret012             Return code = 12
         - - -                    etc...
```

All these instructions need no base register.

37.5.7. Assuming the identifier isn't a negative number, and you write the statement in the form

```
         DC    X'4700nnnn'        Max value is X'7FFF'
```

then its maximum value is $2^{15}-1 = 32767$. But if you write it in the form

```
         NOP   X'nnn'             Max value is X'FFF'
```

then its maximum value is $2^{12}-1 = 4095$.

37.5.8. It works correctly. The MVC instruction puts the return code in the caller's save area where GR15 was originally saved.

37.5.9. Consider these instructions: they use a based branch B, but make no reference to an implicit base-displacement addressing halfword that was assigned by the Assembler.

```
         BASR  14,15              Link to subroutine as usual
         LARL  14,JList           Address of branch list
         B     0(15,14)           Retcode added to list address
JList    J     Ret000             Return code = 0
         J     Ret004             Return code = 4
         J     Ret008             Return code = 8
         J     Ret012             Return code = 12
         - - -
```

37.5.10.

```
         LA    0,3                 Put B'11' in GR0
         NR    0,15                And with 2 low-order bits of RC
         JNZ   BadRC               If nonzero, RC not a multiple of 4
```

37.5.11. The claim is true. Suppose the value in GR15 has nonzero bits a and b in the rightmost two bit positions. Then the ORed expression in the internal register (when the target instruction is built by the CPU) will be

```
         CLI   *+1,B'111111ab'
```

where *+1 is the address of the *unmodified* instruction. Any nonzero value of bits a or b will cause an inequality.

While an ingenious use of CLI and EX, this technique should not be used because it makes a data reference to the instruction stream. It's better to use

```
         TML   15,B'11'
         JNZ   Error
```

Section 37.6

37.6.1. Once the true entry address of the called program is in GR15, we can use it to reference its entry-point ID. This shows possible modifications to Figure 521 on page 784:

```
Caller   STM   14,2,12(13)         Save GR14-GR2 in caller's area
         - - -
         L     15,0(,14)           Load true entry point address
         XR    2,2                 Clear GR2 for EPID length
         IC    2,5(,15)            Get EPID length
         LA    1,6(,15)            Address of EPID string
*        - - -                     Process the EPID somehow
         LM    0,2,20(13)          Restore GR0-GR2
         - - -                     Construct subroutine entry address
         L     14,12(,13)          Restore caller's return address
         BR    15                  Branch to chosen subroutine
```

37.6.2. In this case, we must work back up the save area chain to the caller of this caller's routine, retrieve its entry point address, and use that to find *this* routine's entry-point ID. This shows possible modifications to Figure 521 on page 784:

```
         STM   14,2,12(13)         Save GR14-GR2
         L     14,4(,13)           Back link to our caller's save area
         L     14,16(,14)          Entry address to this routine
         XR    2,2                 Clear GR2 for EPID length
         IC    2,5(,14)            Get EPID length
         LA    1,6(,14)            Address of EPID string
*        - - -                     Process the EPID somehow
         LM    0,2,20(13)          Restore GR0-GR2
         - - -                     Construct subroutine entry address
         L     14,12(,13)          Restore caller's return address
         BR    15                  Branch to chosen subroutine
```

37.6.3. It won't matter, because the c(GR13) are unchanged; but it will waste a few CPU cycles.

37.6.4. You might modify the interface this way:

```
         BAS   14,Caller           Link to caller routine
         NOP   ShftRt#             Pass # of subroutine to be called
         - - -
Caller   AH    15,2(0,14)          Add subroutine number to GR15
         L     15,AdrTbl           Address in GR15 was modified!
         BR    15                  Enter the called routine
AdrTbl   DC    A(ShftRt)           Actual address of 'ShftRt'
         DC    A(Print)            Actual address of 'Print'
         - - -                     Etc., for other assisted routines
```

When control is returned to the instruction addressed by GR14, the NOP will pass control to the following instruction.

You would normally write the linkage routine to provide much more in the way of optional tracing and diagnostic information. The examples given here simply pass control directly to the (indirectly) called program.

37.6.5. The program using assisted linkage to call the **ShftRt** subroutine could use instructions like these:

```
         LHI   0,Print#            Put subroutine number in GR0
         BAS   14,Caller           Link to 'Caller' routine
         - - -                     Return here from called routine
```

The **Caller** routine could be revised like this:

```
Caller   LARL  15,AdrTbl           Address of zero-th target routine
         ALR   15,0                Add subroutine number to GR15
         L     15,0(0,15)          Load correct entry point address
         BR    15                  Enter called routine
AdrTbl   DC    A(ShftRt)           Address of ShftRt
         DC    A(Print)            Address of Print
         - - -                     ...etc...
```

Section 37.7

37.7.1. Because GR14 and GR15 were not modified, and the the result is returned in GR0, we need to restore only GR1-GR3:

```
ShftRt5  STM   14,3,12(13)         Save registers in caller's save area
         LM    2,3,0(1)            Get argument addresses in GR2, GR3
         L     0,0(0,2)            Get 1st (logical) argument in GR0
         L     1,0(0,3)            Get 2nd (integer) argument in GR1
         SRL   0,2                 Shift 2 places
         LTR   1,1                 Test second argument
         JNP   Ret                 Return if it's not positive
         SRL   0,0(1)              Otherwise shift N places
Ret      LM    1,3,24(13)          Restore modified registers
         BR    14                  And return
```

37.7.2. This solution needs to save and restore only GR1; the initial value passed by the caller in GR0 is sometimes called the *seed* of the random number sequence.

```
RanInt   CSect ,
         Using *,15                Establish addressability
         ST    1,SaveR1            Save GR1
         LR    1,0                 Move XOld to GR1
         M     0,A                 Multiply by A
         D     0,P                 Divide Xold*16807 by P
         L     1,SaveR1            Restore GR1
         BR    14                  Return with XNew in GR0
A        DC    F'16807'            Multiplier
P        DC    F'2147483647'       Prime P = 2**31-1
SaveR1   DS    F                   Save area for GR1
         End   ,                   It's a complete program
```

This is a simple example of a very large class of "multiplicative congruential" generators. It has some weaknesses and shouldn't be used for serious simulations.

Programming Problem 37.1.

The internal subroutine is named Print. This (inefficient) solution first prints the even prime 2, and then tests each successive odd number by dividing it by increasing odd values and rejecting the test value if the remainder is zero. Otherwise, the test divisor is squared and compared to the test number; if the square is larger, the test number is prime.

```
P37_1    Start 0
         BASR  15,0                Establish a base register
         Using *,15                Provide addressability
         LA    1,2                 Starting even prime
         JAS   14,Print            Print the only even prime
         LA    11,999              Last value to test
         LA    10,2                Increment 2 for next odd number
         LA    1,3                 Starting odd-number test value
Outer    LA    2,3                 Trial divisor starts at 3
Inner    LR      5,2               Set up multiply
         MR      4,5               Square divisor
         CR      5,1               Compare to test number
         JH      Prime             Branch if the test number is a prime
         LR      5,1               Copy test number
```

```
         SR      4,4              Clear high order register GR4
         DR      4,2              Divide
         LTR     4,4              Test remainder
         JZ      NotPrime         Branch if zero
         AR      2,10             Increment trial divisor by 2
         J     Inner              And try another test value
Prime    JAS   14,Print           Print the prime value
*
NotPrime JXLE  1,10,Outer         Increment by 2, test, and loop
         PrintOut *,Header=NO     Stop
*
Print    CVD   1,Space            Convert to packed decimal
         MVC   Line,Pattern
         ED    Line,Space+6       Maximum of 3 digits in result
         PrintLin Line,L'Line     Print the line
         BR    14                 Return to caller
Line     DC    CL5' '             Output line
Pattern  DC    X'4040202120'      Formatting pattern bbdsd
Space    DS    D                  CVD result work area
         End   P37_1
```

The initial and final output lines look like this:

```
    2
    3
    5
    7
 - - -
  983
  991
  997
```

Programming Problem 37.2.

This little subroutine will do the job very simply:

```
PD2CH    CSect ,
**********************************************************************
* Convert 6-byte packed decimal numbers to numeric characters.
* Calling sequence:
*        CALL PD2CH(packed,character)
* where
*        packed     is a 6-byte packed decimal value
*        character  is a 12-byte area to hold the result characters
**********************************************************************
         USING *,15               Local base register
         STM   14,2,12(13)        Save R14-R2
         LM    1,2,0(1)           Get argument addresses
         MVC   0(12,2),Pattern    Move pattern to output area
         ED    0(12,2),0(1)       Edit the input value
         LM    1,2,24(13)         Restore R1,R2
         BR    14
Pattern  DC    X'40',9X'20',X'2120'  bdddddddddsd for 11 digits
         END
```

Programming Problem 37.4.

This subroutine uses registers 0-4, and restores only registers 2-4.

```
GCD      CSect ,
************************************************************************
* GCD, Greatest-Common-Divisor routine: returns the GCD of two 32-bit *
* positive integers in GR0 or in memory. Calling sequence, using      *
* standard linkage conventions:                                        *
* Call GCD(I,J) returns the result in GR0.                             *
* Call GCD(I,J,K) returns the result at K.                             *
************************************************************************
         STM   2,4,28(13)         Save working registers
         LM    2,4,0(1)           Addresses of I, J, and K
         L     0,0(,3)            Argument J in GR0
         L     2,0(,2)            Argument I in GR2
GCDA     LR    1,2                Copy I to GR1 as first dividend
         LR    2,0                Copy J to GR2 as first divisor
         SR    0,0                Clear high-order register
         DR    0,2                Divide I by J
         CHI   0,1                Is remainder 1?
         JH    GCDA               If greater than 1, try again
         JE    GCDB               Branch if exactly 1
         LR    0,2                If zero, last divisor is the GCD
GCDB     LTR   3,3                Check if J was the last argument
         JM    GCDRet             If so, return with GCD in GR0
         ST    0,0(,4)            Store GCD at K
GCDRet   LM    2,4,28(13)         Restore registers
         BR    14                 Return
         End   ,
```

Section 38 Solutions

Section 38.1

38.1.1. At the point where the Assembler encounters the USING statements, the value of * is 12 larger than the value of **BaseLoc**. (Each LAY instruction is 6 bytes long.)

38.1.2. Because no instructions containing implied addresses appear between the BASR and the USINGs.

38.1.3. This might work if all instructions in the range of the USING statement use 20-bit signed displacements, for which addressability would be available over most of the program. But if statements like

```
         MVC   A,B
A        DS    CL5
B        DC    C'Bytes'
```

appeared anywhere in the first 500K bytes of the program, neither operand of the MVC instruction would be addressable.

38.1.4. The Assembler often places literals at the far end of the program, so they might not be addressable. But even *if* the literal is addressable with a valid base register, there's still an error: the wrong base address is assumed. (See the solution to Exercise 38.1.5.)

38.1.5. Now, at least, the literal is correctly addressable. Registers 3, 4, and 5 will therefore contain the addresses BEGIN+2, BEGIN+4096, and BEGIN+8192. However, the USING promises that they contain BEGIN+2, BEGIN+4098, and BEGIN+8194. Thus, parts of the program will work (those using GR3 as a base register), and other parts mysteriously won't.

38.1.6. This has the same difficulty as in Exercise 38.1.5. The programmer thought he'd get rid of the problems in Exercises 38.1.4 and 38.1.5 by getting rid of the literal.

38.1.7. For the **LA 2,Origin** statement to work correctly, the location named **Origin** must be addressable by some other register. (And if it is, why did we need to use R2 as a base register in the first place?)

38.1.8. There would be no base register available to resolve the implied address **Addrs** in the LM instruction into base-displacement form. The Assembler would indicate an addressability error for that statement. Even if **Addrs** is addressable, all the values promised in the USING statement would then be incorrect.

38.1.9. The Location Counter may not be on an even boundary before the entry point:

```
BigProg  Start 0
         Entry SubRtn
         - - -
         DC    F'9,33'           Two aligned constants
         DC    C'Trouble'        Odd-length constant
         Using *,15              Value of * is now odd!
SubRtn   ST    2,SaveR2          An alignment byte precedes this!
         - - -
```

The USING base address will be too small by 1.

38.1.10. The second USING specifies base addresses that are too large by 4, the length of the LM instruction.

38.1.11.

- The second instruction is BAS, not BASR; it will branch to memory address zero!
- The address promised by the USING statement for the contents of GR10 is Prog+4 (the BAS is four bytes long) so that even if the first error is corrected, the wrong address will be placed in GR11. The generated LA instruction is `41B0AFFC`.

38.1.12. If the literal pool containing `=H'4096'` lies at the end of the assembly (as is often the case), the literal may not be addressable. Even if it *is* addressable, the Assembler will assume GR12 can be used as a base register for the AH instruction, *before* GR12 (at execution time) contains the correct value. Judicious use of LTORG and USING will avoid the difficulty.

38.1.13.

- The USING statement makes no reference to GR10;
- as written, the USING specifies an incorrect value for R11. Even if the USING statement is corrected to read USING *,10,11 we have another error, because

- The USING statement promises that Prog+4098 will be in GR11, not Prog+4096.

38.1.14 Both LA instructions refer to the START statement; they would be assembled as LA 11,4094(,10) and LA 12,4094(,11) which could generate entertaining (or horrifying) results as the program was executed.

38.1.15. You can't define a relocatable symbol on a USING statement! The name-field symbol would be treated as a qualifier on a Labeled USING statement. (There are several other errors that you might want to find, but this one is enough to start with.)

38.1.16. The LA instruction would be assembled as LA 11,0(,11) because the USING statement promises that the address of *+4096 is already in GR11 when the LA is encountered.

38.1.17. The address placed in GR10 by the BASR is `Here+2`, not `Here`. Thus *all* implied addresses will be incorrect when the program is executed. Even if the USING statement is corrected to read `USING Here+2,10,11` then the LA instruction cannot be correct.

38.1.18. This is essentially the same as in Exercise 38.1.16, but with * changed to **`Here`**. GR10 will still contain an undefined value.

38.1.19. By the time the Assembler arrives at the second USING, the LC has increased by 8, so we must write USING *+4088,11 instead.

38.1.20. There's nothing wrong. Base-displacement addresses with GR11 as base may look peculiar, having many odd displacements.*

38.1.21. Yes, so long as HW4096 is (1) on a halfword boundary, as assumed, and (2) is less than 4094 bytes distant from **`BaseLoc`**.

38.1.22. If the DC operand was `H'4000'`, the USING statement would have to be replaced by the three statements

```
Using    *,10
Using    *+4000,11
Using    *+8000,12
```

If the operand was `H'5000'`, a similar set of statements would be needed. However, there would be 904-byte portions of the program that could not be addressed, since the end of the range of GR10's USING statement does not overlap the start of GR11's, and similarly for GR11 and GR12.

38.1.23. First, "AH 11,HW4096" is assembled to `X'4AB0B232'`, and "AH 12,HW4096" is assembled to `X'4AC0B232'`. Second, GR10 would contain `X'00020002'`, and GR11 and GR12 would contain whatever they contained on entry to PROG plus some unknown halfword retrieved from an offset `X'232'` beyond whatever address was in GR11 (which might even have been odd).

When the constant at **`HW4096`** has value `X'2234'`, the two AH instructions are assembled as `X'4AB0C232'` and `X'4AC0C232'`. When executed, GR12 will be used as a base register to find the mystery halfword to be added to GR11 and GR12. If the symbol HW4096 has value `X'3234'`, it is not addressable at all! Thus, if the code in Figure 528 on page 792 is to work, the value of symbol **`HW4096`** must be `X'1000'` or less. (Why not "`X'FFE'` or less"?)

38.1.24. Replace the two `AH reg,HW4096` instructions with `AHI reg,4096` instructions.

38.1.28. Yes, they will work correctly. The only disadvantage is that they will require an additional four bytes.

Section 38.2

38.2.1. The address in GR15 is the address of the entry point **`ShftRt`**, not the address of the instruction following it. Remember that BASR sets the first operand register to contain the address of the instruction *following* the BASR!

38.2.2. Remember that the branch address is formed *before* the branch instruction is executed. Then, the address of the instruction following the BASR (the `ST 0,Result`) is put in the IA portion of the PSW, so the next instruction is fetched from the branch address. The address of the target instruction (at `ShftRt`) will initially be in GR14; after executing the BASR, the next instruction fetched is at `ShftRt`; GR14 contains the *return* address of the ST instruction.

38.2.3. Because the value of * will be wrong: it will be the address of the instruction following the `BASR`, not the address of the entry point! If the `ShftRt` subroutine in Figure 532 on page 798 contained any instructions with implied addresses, they would *not* be resolved relative to the start of the subroutine!

The technique in Figure 532 applies *only* to subroutines without implied addresses, or that establish their own (local) base registers.

---

* To be fair, machine language looks peculiar anyway.

38.2.4. LGFI will set all 64 bits of GG5, whereas L leaves the high-order 32 bits unchanged. Also, LGFI will be be faster.

Section 38.4

38.4.1. Two instances of the literal =F'1' will be generated, both at the end of section AAA, the first control section. The reference to the literal in section BBB will fail with an addressability error.

38.4.2. Two instances of the literal =F'1' will be generated, each following the LTORG instruction at the end of each section.

38.4.3. The END statement ends the assembly, so a DROP isn't needed. But it's *always* a good practice to insert the DROP, because someone else may add more instructions or data to your program.

38.4.4. For this simple little **ShftRt** subroutine, no implied addresses are used (the JNM instruction is a *relative* branch instruction), so no problems will (or should!) occur. But this technique isn't recommended in general.

38.4.5. Suppose someone decides to reorganize the program, and add some more instructions to the **ShftRt** subroutine that require based addresses. It's worth retaining the USING statement in case someone later adds instructions to the routine that will require base-displacement addressing.

38.4.6. The values of the symbols are

```
A      X'001240'
B      X'001248'
C      X'00124C'
D      X'00125C'
N      X'000005'
```

38.4.7. After the 5 (N) bytes at **A** are reserved, the LC value is X'00004B' Because the next statement defines a fullword area of memory, the LC is rounded up to the next fullword boundary, or X'00004C'.

38.4.9. The values of the four symbols are:

```
 A    X'002000'
 B    X'002008'
 C    X'002004'
 D    X'00200C'
```

38.4.10. Consider the terms in the duplication factor expression:

- (*-Prog) is the length of the current section.
- ((*-Prog)+4095): if the section length is not an exact multiple of 4096, this term will have a value exceeding a multiple of 4096.
- (((*-Prog)+4095)/4096*4096): gives the smallest number of 4096-byte "blocks" into which the current section will fit.
- -(*-Prog) subtracts the number of bytes already allocated in the section, leaving the number of bytes in the section needed to extend the section to the next 4096-byte boundary.
- The resulting duplication factor generates the desired number of X'00' bytes (any other constant value could be used).

You should do the above arithmetic steps with LC values like Prog+1, Prog+4095, Prog+4096, and Prog+4097.

38.4.11. The constants will be at these locations:

- CSect A: X'01' at location 0, X'06' at location 1.
- CSect B: X'02,04' at locations X'8-9'.
- CSect C: X'03,05,07' at locations X'10-12'.

38.4.12. The values of the symbols are:

```
 A    X'000003'
 B    X'000005'
 C    X'000001'
 D    X'000004'
 E    X'000006'
 F    X'000002'
 H    X'000000'
```

38.4.13. The values of the symbols are given in the Loc column.

```
 Loc      Source Statement
000000    SectA    CSect ,    Initiate CSECT 'SectA'
000000    H        DS    X    At location 000000
000001    ALoc     LOCTR ,    Start LOCTR group 'ALoc' in 'SectA'
000001    B        DS    X    At location 000001 in 'SectA'
000008    SectB    CSect ,    Initiate CSECT 'SectB'
000008    A        DS    X    At location 000008 in 'SectB'
00000C    BLoc     LOCTR ,    Start LOCTR group 'BLoc' in 'SectB'
00000C    C        DS    X    At location 00000C
000002    ALoc     LOCTR ,    Switch to LOCTR group 'ALoc' in 'SectA'
000002    D        DS    X    At location 000002
00000D    BLoc     LOCTR ,    Switch to LOCTR group 'BLoc' in 'SectB'
00000D    E        DS    X    At location 00000D
000003    SectA    CSect ,    Resume LOCTR group 'ALoc' in CSECT 'SectA'
000003    F        DS    X    At location 000003
00000E    BLoc     LOCTR ,    Switch to LOCTR group 'BLoc' in 'SectB'
00000E    G        DS    X    At location 00000E
000009    SectB    LOCTR ,    Resume LOCTR group 'BLoc' in CSECT 'SectB'
00000A    N        DS    H    At location 00000A
```

38.4.14. When the Assembler processes your END statement, it inserts two invisible statements:

```
<first>  CSECT ,
         LTORG
```

where `<first>` is the name of the first executable control statement.

Section 38.5

38.5.1. The program in Figure 572 on page 825 assembled with the NOTHREAD Assembler option produces this External Symbol Dictionary listing. The only changes from Figure 573 on page 825 are that each control section after the first has starting location zero, and the address of the ENTRY symbol **ASECENT** has been adjusted relative to its owning section's zero start address.

```
                      External Symbol Dictionary

Symbol   Type   Id     Address  Length   Owner Id Flags Alias-of

MAIN      SD 00000001 00002400 000000C0            00
SOMESYM   ER 00000002
ASECTION  SD 00000003 00000000 0000010C            00
ASECENT   LD          00000089          00000003
RSECTION  SD 00000004 00000000 00000095            08
COMSECT   CM 00000005 00000000 00000320            00
ADUMMY    XD 00000006 00000000 0000002C
```

38.5.2. START, CSECT, and RSECT define executable control sections, and COM and DSECT define reference control sections. Only DSECT does not declare space in the object program.

38.5.3. As noted in the statement of the exercise, this type of coding is strongly discouraged!

```
Shifter CSect  ,
        ENTRY  ShftRt,ShfLft      Declare two entry points
        Using  *,15               Set by caller of ShrtRt
ShftRt  MVI    ShftOp,X'88'       Set SRL opcode
        B      ShftAA             Branch to entry code
        Using  *,15               Set by caller of ShfLft
ShfLft  MVI    ShftOp,X'89'       Set SLL opcode
ShftAA  LTR    1,1                Check shift amount
        JNM    ShftOp             Skip if not minus
        SR     1,1                Otherwise set to zero
ShftOp  SLL    0,2(1)         *** Opcode modified to be SLL or SRL
        BR     14                 Return
        End    ,
```

Note that the instructions starting with **ShftAA** do not need a base address, so it doesn't matter that the **Shifter** program can be entered with two different addresses in GR15.

38.5.4. First, some specialized instructions require that their operands be quadword aligned. Second, some instruction sequences will operate more efficiently if their operands are aligned on boundaries that lie within cache lines.

38.5.5. They can be useful for small test programs. Otherwise they mainly exist to let the Assembler generate usable object code in case someone forgot to start a program with a named control section.

38.5.6. You'll generate two unnamed control sections. What your linker does with them should be interesting, depending on how and whether blank section names are also distinguished by type.

38.5.7. You can reference the blank control section by adding an ENTRY statement for the first byte, as in

```
        CSect  ,                  Unnamed control section
        Entry  BlankCS            Entry name
BlankCS DS     XL23
        - - -
```

but you can't define an entry point in a common control section.

38.5.8. If we assume that the address of **ShfLft** is in GR15 on entry at **ShfLft**, the instruction named **ShftRt** is not addressable. However, because there is only a single USING statement, the LA is equivalent to simply LA 15,0(,15). Thus the program will attempt to execute with a value in GR15 that is 8 too large when called at ShfLft.

To correct the program, change LA 15,ShftRt to AHI 15,ShftRt-ShfLft.

38.5.9. Without the third USING statement, displacements will be calculated by the Assembler relative to the address of **ShfLft** as a base, so calls to **ShfLft** will work correctly.

When **ShftRt** is called, only the first two instructions will be executed with the correct base register. Control will be correctly transferred to **ShftAA**, and the LTR will be executed. Now, if the number in GR1 is *not* negative, the branch condition will be met, and the BNM instruction will be executed. However, its addressing halfword is X'F00C'; because GR15 contains the address of **ShftRt**, the branch address is actually the address of **ShftAA**! Thus if c(GR1)≥ 0, the program loops on the two instructions at **ShftAA**.

If c(R1)<0, the BNM instruction is not executed, and control passes to **ShftOK**. The bit tested will actually be at ShFlag-8, which is the first byte of the "BR 14" instruction! This byte contains X'07', so the tested bit will be one and the branch condition for the following BZ will not be met. The operand will be shifted *left* instead of *right!*

This exercise shows why you must be careful in establishing correct USING statements in routines with multiple entry points.

38.5.10.

```
BYTE     Start 0                     Conversion subroutine
         Using *,15                  Caller provides GR15 base register
         STM   14,2,12(13)           Save GR14-GR2
         LM    1,2,0(1)              Load pointers to the two arguments
         ICM   1,B'1000',0(1)        Insert the byte argument
         LA    0,8                   Initialize bit counter
         MVC   0(8,2),ZChars         Move zero-characters to argument 2
Loop     LTR   1,1                   Test sign bit of GR1
         JNM   Next                  Branch if it's zero
         OI    0(2),X'01'            Make the output character a '1'
Next     LA    2,1(,2)               Increment output-character address
         ALR   1,1                   Shift the byte left by 1 bit
         JCT   0,Loop                Repeat for all 8 bits
         LM    0,2,20(13)            Restore GR0-GR2
         BR    14                    Return to caller
ZChars   DC    8C'0'                 Eight zero characters
         End
```

38.5.11. Consider this subroutine: no implicit addresses are used, so there's no need for the normal Using *,15 statement.

```
SIGNUM   Start 0                  Start of SIGN control section
         STM   14,1,12(13)        Save registers
         SR    0,0                Initialize result
         L     1,0(,1)            Get address of the argument
         LT    1,0(,1)            Get the integer argument, set CC
         L     1,24(,13)          Restore GR1 without changing CC
         BZR   14                 Return directly if argument = 0
         BCTR  0,0                Make c(GR0) = -1 (no change to CC)
         BMR   14                 And return
         LCR   0,0                Make c(GR0) = +1
         BR    14                 Return to caller
         End
```

Note that the BCTR instruction can't be changed to `AHI 0,-1` because AHI changes the condition code.

38.5.12. Because **Data** is an external symbol, and is not addressable.

38.5.13. This subroutine assumes that the **MemDump** subroutine uses a local register save area so that it won't modify the contents of the caller's save area (which may be in an area to be dumped).

```
MemDump  Start 0                  Memory-dump subroutine
         Using *,15               Establish addressability
         STM   1,5,Save           Save modified registers
         LM    1,2,0(14)          Load start and end addresses
MainLoop MVC   Line,CC            Set carriage control character
         ST    1,Loc              Store starting address for this line
         MVC   Loc+4(16),0(1)     Move 16 bytes for unpacking/formatting
         LA    5,Loc              Initialize UNPK source address
         LA    3,5                Format 5 words into printable hex
         LA    4,Line             Initialize output line position
LineLoop UNPK  0(9,4),0(5,5)      Unpack a word
         TR    0(8,4),TRTab       Translate to EBCDIC
         MVI   8(4),C' '          Blank out the swap byte
         LA    5,4(,5)            Increment input address by 4
         LA    4,9(,4)            Increment output address by 9 (space!)
         JCT   3,LineLoop         Finish one line of output
         PrintLin CC,1+L'Line     Print the line
         LA    1,16(,1)           Increment starting address by 16
         CR    1,2                Compare to end address
         JNH   MainLoop           Repeat for another line
         LM    1,5,Save
         B     8(,14)             Return to caller
Save     DS    5F                 Save area
Loc      DS    5F                 Work area for address, dump words
CC       DC    C' '               Blank carriage control character
Line     DS    CL45               Output line
         DS    X                  Padding byte for UNPK byte swap
TRTab    DC    240C' ',C'0123456789ABCDEF'  Translate table
         End
```

38.5.15. This can be done in many ways; first, we illustrate a technique used before z/Architecture instructions were available.

```
I2D      CSect ,                  Word integer to long hex float
         Using *,15               Assume standard entry-address reg
         ST    0,Float+4          Store integer value
         XI    Float+4,X'80'      Invert sign bit only
         LD    0,Float            Pick up integer and exponent
         SD    0,Const            Subtract magic constant
         BR    14                 Return
Const    DC    DS6'2147483648'    Scaled 2**31
Float    DC    X'4E',7X'0'        Temporary with hex exponent 14
```

This solution uses z/Architecture instructions:

```
I2D      CSect ,                  Word integer to long hex float
         Using *,15               Assume standard entry-address reg
         LGFR  0,0                Extend 32-bit operand to 64 bits
         LDGR  0,0                Copy GG0 to FPR0
         LNDR  0,0                Invert sign bit of FPR0
         SD    0,Const            Subtract magic constant
         BR    14                 Return
Const    DC    DS6'2147483648'    Scaled 2**31
```

38.5.18. Here is an example of a calling sequence:

```
         LA    1,ArgList          Point o argument addresses
         L     15,=V(ShftRt)      Load subroutine address
         BASR  14,15              Call it
         - - -
ArgList  DC    A(Logic,NN,Result) Argument addresses
Logic    DS    F                  Bits to be shifted
NN       DS    F                  Shift amount
Result   DS    F                  Shifted quantity
```

Section 38.6

38.6.2. Both generate an ER entry for X in the ESD record, but the V-type constant generates a different RLD entry that lets the Linker assign an indirect resolution to X.

Section 38.7.

38.7.1. An ENTRY name always has the same “owning” ESDID as its containing control section, but does not have an ESDID of its own. (That's why the * is attached to the ESDID.)

38.7.2. This can be done by using the fact that the owner of a COM section can refer to its internal names as ordinary symbols. Thus, the owner of the **MyCom** section can define an address constant pointing to the symbol **YourData** and give the adcon a name like **DataAddr** declared in an ENTRY statement:

```
Owner    CSect ,                  Owner of the MyCom common section
         - - -
DataAddr DC    A(YourData)        Address of the data area
         ENTRY DataAddr           Define its name externally
         - - -
MyCom    COM   ,                  The Owner's common section
         - - -                    Parts of the MyCom section
YourData DS    XL(23456)          The data area you need to reference
         - - -                    Other parts of the MyCom section
```

Then, in your program you can write

```
YourProg CSect .                  Program to reference YourData
         EXTRN DataAddr           Name of the YourData pointer
         - - -
         L     3,DataPtr          Get the Owner's 'YourData' pointer
         L     3,0(,3)            Get the YourData pointer
         - - -                    ... and work with the data
DataPtr  DC    A(DataAddr)        Point to MyCom's YourData pointer
```

38.7.3. Use the contents of the Address as both an addend and as a mask to round up the current offset to the required boundary. Suppose the current offset is in register ROff::

```
         L     0,Addr             Get the XD item's Address field
         ALR   ROff,0             Add to the current offset
         NR    ROff,0             Set low-order bits to zero
```

38.7.4. If there were more than 1024 PR items each having length 4, the offset in the QL2-con would have a nonzero high-order digit. That digit would appear in the base-register digit position of the base-displacement field of the L instruction.

Section 38.10

38.10.5. Since the four instructions all set bit 32 of the R_2 operand register (the "**b**" bit) to 0 in 24-bit mode and to 1 in 31-bit mode, these are the settings that BSM uses to set the AMODE before branching. It will set the PSW AMODE bits to the same value they had at the time of the call.

38.10.6. BSM: yes; BASSM: no, because the first operand register is always changed in ways that depend on the current addressing mode.

38.10.7. Remember that BAL is a 4-byte instruction that sets the ILC to `B'10'`, which means the high-order bit of the R_1 register will be 1. If your program runs in 24-bit addressing mode and you try to return from the called routine with BSM, it will set the addressing mode to 31, and any address with nonzero bits in positions 32-40 will try to address something beyond the bounds of your program.

Note also that executing BAL or BALR with either EX or EXRL instructions will set the high-order bit of the R_1 register.

Executing in AMODE 31 is safe for calling another AMODE 31 program, because both BAL and BALR set the high-order bit of the 32-bit return address to 1; returning with BSM doesn't change the addressing mode.

38.10.8. It would work, but you'd have to set the appropriate bits in the R_1 and R_2 registers, which isn't necessary for the "ordinary" linkage instructions that don't change addressing mode.

38.10.10. The routine D could have had any of three RMODE settings:

RMODE 24 D will be loaded below the 16MB "line".

RMODE 31 D will be loaded above the 16MB "line" if **C** and **D** are in separate classes in a program object. If they are both components of a load module, they both will be loaded below the 16MB "line", because load modules are given the lowest RMode of all its components.

RMODE ANY D could be loaded either below or above the 16MB "line", probably above, again depending on the type of executable created by the Linker.

Programming Problem 38.2.

The `Zeller` subroutine has three arguments, the year, month, and day of month. Note that this solution makes no local storage references, and therefore has no USING statement!

```
Zeller   CSect ,                  No base register needed!
* Zeller's Congruence for the day of the week (0=Saturday, 6=Friday).
* D_W = (D_M+Y+Y/4+6*(Y/100)+Y/400+((M+1)*26)/10) (mod 7).
*        All divisions truncate, and return floor(quotient)
         STM   14,5,12(13)        Save registers
         LM    1,3,0(1)           Addr(Year,Month,Day_of_Month)
         L     5,0(,1)            Get Y (year)
         L     1,0(,3)            Get D_M (day of month)
         AR    1,5                D_M+Y
         SRA   5,2                floor(Y/4)
         AR    1,5                D_M+Y+floor(Y/4)
         LA    3,25               Constant 25
         XR    4,4                Clear high-order register
         DR    4,3                floor(Y/100)
         LR    4,5                Copy to GR4
         SRA   4,2                floor(Y/400)
         AR    1,4                D_M+Y+floor(Y/4)+floor(Y/400)
         MHI   5,6                6*floor(Y/100)
         AR    1,5                D_M+Y+fl(Y/4)+fl(Y/400)+6*fl(Y/100)
         LA    3,5                Constant 5
         L     5,0(,2)            Get M (month)
         AHI   5,1                M+1
         MHI   5,13               (M+1)*13
         XR    4,4                Clear high-order register
         DR    4,3                ((M+1)*13)/5
         AR    1,5                Add to sum
         LA    4,7                Constant 7
         XR    0,0                Clear high-order register
         DR    0,4                Divide by 7, leave remainder in GR0
         LM    1,5,24(13)         Restore all but GR0
```

```
         BR    14
         End
```

The calling program could be written like this:

```
P38_2    CSect ,
         Print NoGen
         STM   14,12,12(13)       Save registers
         LR    2,13               Copy address of caller's area
         CNOP  0,4                Align to 4-byte boundary
         BRAS  13,EndSave         Set GR13 to address of our save area
         Using Save,13            Use GR13 as local base register
Save     DC    18F'0'             Local save area
EndSave  ST    2,4(,13)           Store back chain in our area
         ST    13,8(,2)           Put forward chain in caller's area
*
Read     ReadCard  Inrec,NoData   Read an input record
         MVC   CYear,InRec+6      Move year characters for output
         MVC   CDay,InRec+28      Move day characters for output
         PACK  DWork,Inrec(10)    Convert Year to packed decimal
         CVB   0,DWork            Convert to binary
         ST    0,Year             Save the binary year value
         PACK  DWork,Inrec+10(10) Convert Month to packed decimal
         CVB   0,DWork            Convert to binary
         ST    0,Month            Save the binary month value
         PACK  DWork,Inrec+20(10) Convert Day of month to packed dec
         CVB   0,DWork            Convert to binary
         ST    0,DayofM           Save the binary day of month value
         MVI   Line,C' '          Initialize output line to blanks
         MVC   Line+1(L'Line-1),Line   Ripple move
*
         LA    1,ArgList          Point to argument address list
         L     15,AZeller         Get address of Zeller routine
         BASR  14,15              Call Mr. Zeller for his congruence
         ST    0,WeekDay          Store the result
*                                 Format the data
         LR    1,0                Copy to GR1
         AR    1,1                Double it for index
         LA    3,Line+1           Point to initial character
         XR    2,2                Clear GR2 for EX lengths
         IC    2,Days+1(1)        Get effective length of day name
         IC    1,Days(1)          Get offset of day name
         LA    1,DCh(1)           Point to day name
         EX    2,MoveDay          Move day name to output line
         LA    3,2(3,2)           Step output address over day
         L     1,Month            Get month value (starts at 1)
         AR    1,1                Double for indexing
         IC    2,Months-1(1)      Get effective length of month name
         IC    1,Months-2(1)      Get offset of month name
         LA    1,MCh(1)           Point to month name
         EX    2,MoveDay          Move month name to line
         LA    3,2(2,3)           Step over month name
         MVC   0(2,3),CDay        Move character form of day
         CLI   0(3),C' '          Was leading character a blank?
         JNE   DoYear             Skip adjustment if yes
         MVC   0(1,3),CDay+1      Move the single-digit day
         BCTR  3,0                Back up by 1 character
DoYear   MVI   2(3),C','          Insert comma
         MVC   4(4,3),CYear       Insert year characters
         PrintLin Line,L'Line     Print the result
         J     Read               Read another input record
NoData   PrintOut *,Header=NO
MoveDay  MVC   0(*-*,3),0(1)      Move day character name to line
*
AZeller  DC    V(Zeller)          Address of Zeller subroutine
ArgList  DC    A(Year,Month,DayofM)    Argument addresses
```

```
WeekDay  DS    F
DayofM   DS    F
Month    DS    F
Year     DS    F
DWork    DS    D                   Doubleword work area
CYear    DS    CL4
CDay     DS    CL2
*
Days     DC    AL1(DSat-DCh,L'DSat-1)
         DC    AL1(DSun-DCh,L'DSun-1)
         DC    AL1(DMon-DCh,L'DMon-1)
         DC    AL1(DTue-DCh,L'DTue-1)
         DC    AL1(DWed-DCh,L'DWed-1)
         DC    AL1(DThu-DCh,L'DThu-1)
         DC    AL1(DFri-DCh,L'DFri-1)
DCh      DS    0C
DSat     DC    C'Saturday'
DSun     DC    C'Sunday'
DMon     DC    C'Monday'
DTue     DC    C'Tuesday'
DWed     DC    C'Wednesday'
DThu     DC    C'Thursday'
DFri     DC    C'Friday'
*
Months   DC    AL1(MJan-MCh,L'MJan-1)
         DC    AL1(MFeb-MCh,L'MFeb-1)
         DC    AL1(MMar-MCh,L'MMar-1)
         DC    AL1(MApr-MCh,L'MApr-1)
         DC    AL1(MMay-MCh,L'MMay-1)
         DC    AL1(MJun-MCh,L'MJun-1)
         DC    AL1(MJul-MCh,L'MJul-1)
         DC    AL1(MAug-MCh,L'MAug-1)
         DC    AL1(MSep-MCh,L'MSep-1)
         DC    AL1(MOct-MCh,L'MOct-1)
         DC    AL1(MNov-MCh,L'MNov-1)
         DC    AL1(MDec-MCh,L'MDec-1)
MCh      DS    0C
MJan     DC    C'January'
MFeb     DC    C'February'
MMar     DC    C'March'
MApr     DC    C'April'
MMay     DC    C'May'
MJun     DC    C'June'
MJul     DC    C'July'
MAug     DC    C'August'
MSep     DC    C'September'
MOct     DC    C'October'
MNov     DC    C'November'
MDec     DC    C'December'
*
Line     DC    CL40' '
InRec    DS    CL80                Input record buffer
         End   P38_2
```

The printed output from the suggested test cases is:

```
 Tuesday June 30, 2009
 Friday January 1, 2000
 Saturday January 1, 1900
```

Programming Problem 38.3. Note that this solution modifies none of the registers that must be restored on return.

```
HexUlp   CSect ,                  Return ulp of hex float arguments
*        Calling sequences:
*        Call  HxUlpE(X)          Calculate ulp(X) for short X
*        Call  HxUlpD(X)          Calculate ulp(X) for long X
*        Call  HxUlpL(X)          Calculate ulp(X) for extended X
*        ulp(x) is returned in FPR0 (for short and long X), and in
*        FPR0/2 for extended X.
         ENTRY HxUlpE,HxUlpD,HxUlpL
*
HxUlpE   L     1,0(,1)            Get address of argument X
         XGR   0,0                Clear GG0
         OIHL  0,1                Set bit 31 to 1
         ICMH  0,8,0(1)           Insert characteristic 'cc' of X
         LFGR  0,0                Copy X'cc000001 00000000' to FPR0
         DE    0,0(,1)            Divide by X
         BR    14                 Return short ulp(X) to caller
*
HxUlpD   L     1,0(,1)            Get address of argument X
         XGR   0,0                Clear GG0
         OILL  0,1                Set bit 63 to 1
         ICMH  0,8,0(1)           Insert characteristic 'cc' of X
         LFGR  0,0                Copy X'cc000000 00000001' to FPR0
         DD    0,0(,1)            Divide by X
         BR    14                 Return long ulp(X) to caller
*
HxUlpL   L     1,0(,1)            Get address of argument X
         XGR   0,0                Clear GG0
         LD    4,0(,1)            Put high half of X in FPR4
         LD    6,8(,1)            Put low half of X in FPR6
         ICMH  0,8,0(1)           Insert characteristic 'cc' of X
         LFGR  0,0                Copy X'cc000000 00000000' to FPR0
         LGHI  0,1                Put X'00000000 00000001' in GG0
         LFGR  2,0                Copy it to FPR2
         DXR   0,4                Divide by X
         BR    14                 Return extended ulp(X) to caller
         End
```

Programming Problem 38.9. The changes of addressing mode in this little program are needed because the `PrintOut` macro operates only in AMODE(24):

```
AM   CSect ,
     Using *,15
     LG    0,=FD'-1'        Preset GG0
     LGR   1,0              Copy to GG1
     SAM24 ,                Set addressing mode 24
     BASSM 1,0              BASSM sets GG1
     LGR   2,0              Copy GG0 to GG2
     SAM31 ,                Set AM31
     BASSM 2,0              BASSM sets GG2
     LGR   3,0              Copy GG0 to GG3
     SAM64 ,                Set AM64
     BASSM 3,0              BASSM sets GG3
     SAM24 ,                Reset to AM24 for PRINTOUT
     PrintOut 16,17,18,19,Header=No    Display the results
*
     LGR   4,0              Copy GG0 to GG4
     SAM24 ,                Set addressing mode 24
     BASR  4,0              BASR sets GR4
     LGR   5,0              Copy GG0 to GG5
     SAM31 ,                Set AM31
     BASR  5,0              BASR sets GR5
     LGR   6,0              Copy GG0 to GG6
     SAM64 ,                Set AM64
     BASR  6,0              BASR sets GG5
     SAM24 ,                Reset to AM24 for PRINTOUT
```

```
     PrintOut 20,21,22,*,Header=No
     End
```

The results look like this (explanations added):

```
 GGR   0  = X'FFFFFFFFFFFFFFFF'   (Initialization value)

 GGR   1  = X'FFFFFFFF0002100E'   (AM24: b bit 32 = 0)
 GGR   2  = X'FFFFFFFF80021016'   (AM31: b bit 32 = 1)
 GGR   3  = X'000000000002101F'   (AM64: e bit 63 = 1)

 GGR   4  = X'FFFFFFFF0002107C'   (AM24: b bit 32 = 0)
 GGR   5  = X'FFFFFFFF80021084'   (AM31: b bit 32 = 1)
 GGR   6  = X'000000000002108C'   (AM64: e bit 63 = 0)
```

so the first-operand mode-bit settings are the same in AM24 and AM31.

Programming Problem 38.10.

```
HexExp   CSect ,                  Return exponent of HFP argument
         L     1,0(,1)            Get argument address
         XR    0,0                Clear GR0
         ICM   0,1,0(1)           Insert sign and characteristic
         NILL  0,B'01111111'      Eliminate sign bit
         AHI   0,-64              Remove bias
         BR    14                 Return to caller
         End
```

Section 39 Solutions

Section 39.1

39.1.1. The symbols and their values are:

```
Value      Symbol

000000     D39_1_1  DSect ,
000000     A        DS    CL9
00000C     B        DS    F
000010     C        DS    X
000012     D        DS    H
000018     E        DS    D
000020     F        Equ   *-D39_1_1
```

39.1.2. Using the same symbols as in Figure 57 on page 161, we can define the DSECT this way:

```
ALStmt   DSect ,                Assembler Language statement
Statemnt DS    0CL80            Define 80-column record area
Name     DS    0CL8             Define name-field symbol
         DS    CL9              Reserve space for name-field symbol
Mnemonic DS    0CL5             Define 5-character mnemonic field
Mnemopnd DS    0CL25            Define both mnemonic and operands
         DS    CL6              Reserve space for mnemonic
Operand  DS    0CL19            Define 19-character operand field
         DS    CL20             Reserve space for operand field
Comment  DS    CL36             Allocate 36 columns for comments
Continue DS    C                Define continuation-indicator column
Sequence DS    CL8              Define sequencing columns
ALStmt_L Equ   *-ALStmt         Length of the DSECt
```

39.1.3. As defined in the solution to Exercise 39.1.2, the values and length attributes of all symbols are:

Symbol	Value (Hex)	Length Attr.
Statement	000000	80
Name	000000	8
Mnemonic	000009	5
MnemOpnd	000009	25
Operand	00000F	19
Comment	000023	36
Continue	000047	1
Sequence	000048	8
ALStmt_L	000050	1

Section 39.2

39.2.1. Following the Equ statement defining the value of the symbol RecLen, add these two statements:

```
         Org   Record           Position LC at start of DSECT
RecBase  DS    XL(RecLen)       Define symbol with record length
```

39.2.2. You can do this with a single statement:

```
NewRec   DS    0D,XL(RecLen)
```

Section 39.3

39.3.1. We assumed in Figure 625 on page 876 that the offset of the symbol **RecID** in the **Record** DSECT is greater than 4096. So that the displacement of the second operand will be less than 4096, we must subtract 4096 and assign the base register (GR8) with contents 4096 greater than the contents of GR7. You should never program this way!

39.3.2. The generated instruction is X'D204 000A 700A'. The base register for the first operand is zero, so the target address is at address X'00000A', very probably causing a protection exception. (Note also that the Effective Length is 4, meaning that the MVC is attempting to move 5 bytes.)

39.3.3. The generated instruction is X'D203 500A 700A'. Both the target address and the Effective Length are correct.

39.3.4. The LAY instruction (assuming the `USING 0+X'F999',7` statement in the footnote on page 879) will generate `E320 7D07 0871`, with GR7 as base register. (Note that the base value `X'F999'` plus the displacement `X'8D07'` is `X'186A0'`=100000.)

Section 39.4

39.4.1. The instructions and their generated object code are:

```
         LA    1,ABC              4110 C008
         LA    2,Qual.ABC         4120 9004
```

39.4.2. The instructions and their generated object code are:

```
Qual     Using *,9
         Using *,6
         LA    1,ABC              4110 6008
         LA    2,Qual.ABC         4120 9008
```

Even though both instructions have the same displacement, the first USING specifies the higher-numbered register, so it cannot be used to resolve the unqualified implied address of the first LA instruction. The lower-numbered register, GR6, is assigned as base.

39.4.3. The three instructions and their generated object code are:

```
  4100 9028             LA    0,A+40      Ordinary USING
  4110 9028             LA    1,QQ.A+40   Labeled USING
  4120 9028             LA    2,QQ.A+40   Labeled USING
```

All three generate the same object code.

39.4.4. You might expect the generated object code to be `X'4130 7200'`. However, the Assembler doesn't resolve qualified absolute symbols, so the statement generates a diagnostic.

Section 39.5

39.5.2. The three DSECTs as written are short enough that all three can be addressed with a single base register, so the order of their mapping with the Dependent USING statements does not matter. However, if one of the DSECTs is very long (for example, DSECT A is defined with a `DS  XL6000` statement), it might be best to map the three DSECTS with the longest one last.

39.5.3. If the two CSECTs are not assembled together, none of the symbols in the `MsgSkels` CSECT would be known in the `Program` CSECT.

39.5.4. Consider these DSECT definitions:

```
A     DSect ,             B     DSect ,         C     DSect ,
      DS    CL32          B1    DSD             C1    DS    CL80
BLoc  DS    XL(Blen)      B2    DS    D         C2    DS    XL8
CLoc  DS    XL(Clen)      BLen  Equ   *-B       CLen  Equ   *-C
      DS    (256-(*-A))
```

Section 39.6

39.6.2. Assuming that GR9 provides addressability to the `Outer` DSECT, you could write something like this:

```
         Using Outer,9
         LA    10,Out_Inr1        A(Out_Inr1) in GR10
         Using Inner,10           Addressability for first instance
         LA    11,Out_Inr2        A(Out_Inr1) in GR11
         Using Inner,11           Addressability for second instance
```

Three base registers are needed, rather than one.

39.6.3. The `Top` DSECT starts at location zero. The length of the `Bot` DSECT is 16 (`X'10'`) bytes because it is rounded to the next doubleword boundary. The `Mid` DSECT length is `X'C8'` bytes. The `Top` is three times as long as `Mid`, or `X'258'` bytes. Offsets of the other variables within their respective DSECTs are easy to determine. The offsets of the four variables are:

- `M1B1.X1        X'028'`
- `Mid1.MidVar1   X'038'`
- `M3B2.X2        X'209'`
- `Mid3.MidVar4   X'238'`

39.6.4. The generated instructions are:

```
Object Code          Instruction
D204 7028 702D       MVC   M1B1.X1,M1B1.X2
D204 709D 7028       MVC   M1B3.X2,M1B1.X1
D204 7209 722D       MVC   M3B2.X2,M3B3.X2
D204 70F0 7209       MVC   M2B1.X1,M3B2.X2
D20F 71B8 7228       MVC   Mid3.B1,Mid3.B3
D20F 7098 70F0       MVC   Mid1.B3,Mid2.B1
D213 714C 7190       MVC   Mid2.MidVar3,Mid3.MidVar1
D2C7 7000 70C8       MVC   M1,M2
```

Section 39.7

39.7.1. The DSECTs and their lengths are:

```
Date        X'008'  =   8
Addr        X'050'  =  80
Phone       X'010'  =  16
```

39.7.2. The DSECTs and their lengths are:

```
Person      X'098'  = 152
Employee    X'373'  = 883
```

39.7.5. Yes.

Section 40 Solutions

Section 40.1

40.1.1. The three statements should be written

```
LowerSub DC    A(LBound)
UpperSub DC    A(HBound)
         - - -
XX       DC    (HBound-LBound+1)F'12345'
```

40.1.2. Yes, it is open to criticism on many grounds.[337]

1. Consider what happens if the addressing halfword of the L instruction named **GetIt** could address the start of the table, but not the end.
2. Suppose the initial addressing halfword is `X'FFF0'`: what happens when we want to fetch the fifth table element?
3. The method is quite inefficient, especially considering the availability of many instructions to do indexing automatically,
4. Modifying instructions is considered an extremely poor technique under any circumstance: you can't debug the code by reading the source or the listing, and the code cannot be shared by more than one process.
5. It's terribly inefficient, because storing into the instruction stream makes the CPU stop pre-processing instructions and start over.

40.1.3. You can make a good case for either view:

- In favor of spacing by 8: Each element of the table is aligned on a halfword boundary, making the halfword integer more likely to be aligned on correct boundaries; and calculating index offsets can use shift logical instructions instead of multiplying by 7.
- Not in favor of spacing by 8: In programs needing to conserve memory space, adding the extra byte to each entry could cause addressability problems.

In general, speed of access and simplicity of indexing code are more important considerations.

40.1.4. Only the single element of **XX** whose index is found at **LowerSub** will be stored at **XSum**.

40.1.5. Consider writing the DSECT this way:

```
Elem     DSect ,
El_Int   DS    HL2
El_Chars DS    CL5
```

The reason for writing the operand of `El_Int` as `HL2` rather than `H` is to remind the reader that the integer field may not always be aligned on a halfword boundary.

Section 40.2

40.2.1. To answer the last two questions first:

- The ordering of the array does not matter. Suppose the elements are stored in row order. Then

  ```
  Addr(A(i,j)) = Addr(A) + L*[N*(i-1)+(j-1)]   ('from' element)
  Addr(A(j,i)) = Addr(A) + L*[N*(j-1)+(i-1)]   ('to' element)
  ```

  If the elements are stored in column order,

  ```
  Addr(A(i,j)) = Addr(A) + L*[(i-1)+N*(j-1)]   ('from' element)
  Addr(A(j,i)) = Addr(A) + L*[(j-1)+N*(i-1)]   ('to' element)
  ```

  Since the array A is square, and we are simply exchanging elements, there is no difference between row and column order.
- We don't have to swap the N "diagonal" elements A(i,i), so the number of elements involved is $N*N-N$, or $N*(N-1)$. The elements are swapped in pairs, so the number of swaps is $N*(N-1)/2$.

The code will need to do swaps only for subscripts $j<i$, for values of i from 2 to N and for values of j from 1 to $i-1$. (The inner loop is indented to help readability.)

[337] Fortunately, that programmer has been out of business for many years.

```
RI       Equu  2                   Register for i values
RiMax    Equ   3                   Register for maximum i value
RJ       Equ   4                   Register for j values
RjMax    Equ   5                   Register for maximum j value
         LA    RiMax,N             Maximum value of i
         LA    RI,2                Initial value of i
Outer    LR    RjMax,RI            Copy current value of i
         BCTR  RjMax,0             Maximum value of j = i-1
         LA    RJ,1                Initial value of j
Inner      LR    8,RI                Copy i
           BCTR  8,0                 Form (i-1)
           LR    10,8                Copy (i-1) to GR10
           LR    9,RJ                Copy j
           BCTR  9,0                 Form (j-1)
           LR    11,9                Copy (j-1) to GR11
           MHI   9,N                 Form N*(j-1)
           SLDL  8,2                 Form L*(i-1) and L*N*(j-1)
           ALR   8,9                 For linear subscript for A(i,j)
           MHI   10,N                Form N*(i-1)
           SLDL  10,2                Form L*N*(i-1) and L*(j-1)
           ALR   10,11               For linear subscript for A(j,i)
             L     0,Array(8)          Get Array(i,j) for swap
             L     1,Array(10)         Get Array(j,i) for swap
             ST    0,Array(10)         Store Array(i,j) at old A(j,i)
             ST    1,Array(8)          Store Array(j,i) at old A(i,j)
           AHI   RJ,1                Increase j by 1
           CR    RJ,RjMax            See if we've completed a j cycle
           JNH   Inner               If not, repeat
         AHI   RI,1                Increment i by 1
         CR    RI,RiMax            Compare to upper limit for i
         JNH   Outer               Repeat another cycle on i
         - - -
Array    DS    (N*N)F              N by N array
```

40.2.3. In a column-ordered array with R rows, we know that the address of an element A(i,j) is given by

`addr(A(i,j)) = [addr(A(1,1))-(L*(R+1))]+[L*(i+R*j)].`

We know the address of A(i,j); so after rearranging, we find

`(i-1)+R*(j-1) = [addr(A(i,j))-addr(A(1,1))]/L`

If we divide the right-hand expression by R, the remainder is (i-1) and the quotient is (j-1).

```
L        Equ   4                   Assume elements are 4 bytes long
R        Equ   7                   Assume 7 rows
         L     1,ElemAddr          Address of A(i,j)
         S     1,BaseAddr          Subtract address of A(1,1)
         SRL   1,2                 Divide by L
         SR    0,0                 Clear high-order register for divide
         D     0,=A(R)             Divide by number of rows
         AHI   0,1                 Add 1 to remainder (i-1)
         ST    0,ISub              Store I subscript
         AHI   1,1                 Add 1 to quotient (j-1)
         ST    1,JSub              Store J subscript
         - - -
ElemAddr DS    A                   Known address of A(i,j)
BaseAddr DS    A                   Known address of A(1,1)
ISub     DS    F                   Calculated i subscript
JSub     DS    F                   Calculated j subscript
```

Section 40.3

40.3.7. We simply exchange the corresponding elements of the `IVal` and `JVal` arrays. These instructions will transpose the sparse matrix A:

```
         L     1,NbrEls          Get number of array elements
         BCTR  1,0               Decrement by 1
         SLA   1,2               Make into a word index
         JNP   Done              Exit if no elements today
         LA    0,4               Set word increment
         SR    0,0               Initialize index
Next     L     3,IVal(2)         Get an I subscript
         L     4,JVal(2)         And a J subscript
         ST    3,JVal(2)         Store I where J used to be
         ST    4,Ival(2)         Store J where I used to be
         JXLE  2,0,Next          Increment index, test, and loop
Done     - - -                   Transposition complete
```

Section 40.4

40.4.6. Here's one solution:

```
AddrTab  DC    (NRows)A(AA-L+L*(NRows*(*-AddrTab)/4))
```

40.4.7. There are these factors to consider:

1. Subscript sizes must be small enough so that address calculations will not overflow.
2. If subscript sizes are limited to avoid overflows, the address table may be so large that the additional memory needed (in addition to the space required for the array itself) that it may be simpler to calculate "normal" subscript addressing.
3. There is a fixed amount of storage available between the 16MB "line" and the 2GB "bar" (about 2^{29} bytes).

Section 40.5

40.5.1. It will not matter, because you will be searching through the entire array.

40.5.2. Unless we were lucky enough to find the desired item on the first comparison, the search would end with the element not found. For example, if the array elements were 5, 4, 3, 2, and 1 and we search for the value 2, the first comparison (2 to 3) would show that we should move lower in the array. But all those elements are larger than 2, so the search would never find the correct element.

40.5.5. You won't see much improvement, because the effort of comparing the sign of the search argument to the sign of the current table "probe" element is about the same as just comparing the argument to the table entry.

40.5.7. The best estimate is N*(N-1)/2 comparisons.

Section 40.6

40.6.1. (3) will overwrite the element currently at the top of the stack, and then set the stack pointer to the address of an empty stack position.

40.6.2. The LM/STM/LA instructions will do the initialization:

```
SP       Equ   1                 GR1 contains stack pointer
StkSize  Equ   20                Number of elements for full stack
         DS    (StkSize)F        Allocate space for the stack
Stack    DS    0F                Define name of stack area
         - - -
         LM    0,2,=F'6,2,9'
         STM   0,2,Stack-3*L'STACK     Set values
         LA    SP,Stack-3*L'STACK      Initialize stack pointer
```

40.6.3. The main disadvantage of this method is that a check for stack overflow might be done after the "push", which might be too late. One (very) small advantage is that a CPU that overlaps the execution of several instructions need not wait for the address of the stack top to be generated before it can use that address to store the new data item.

40.6.4. The values of the expressions are 210, 77, and 100.

(1) A possible infix notation is 2*3*5*7

```
Operator                *   *   *
Operand    7   3   2   5  10  30  210
  Stack        7   3   2   3   7
                   7   3   7
                       7
```

(2) A possible infix notation is `7*((3*2)+5)`

```
Operator            *       +   *
Operand    7   3   2   6   5  11  77
  Stack        7   3   7   6   7
                   7       7
```

(3) A possible infix notation is `(3+7)*(2*5)`

```
Operator         +          *   *
Operand    7   3  10   2   5  10  100
  Stack        7      10   2  10
                          10
```

40.6.5. The first 5 statements define some symbolic names for registers.

```
EP       Equ   11                  Expression scan pointer
SP       Equ   10                  Stack pointer
StkSize  Equ   10                  Stack size
WT       Equ   8                   Work reg for type codes
WV       Equ   9                   Work reg for values, operators

         SR    SP,SP               Initialize SP to 'empty stack'
         LA    EP,Expressn         Initialize expression scan pointer
Fetch    LH    WT,0(,EP)           Get expression item type code
         LH    WV,2(,EP)           Get operand or operator
         LTR   WT,WT               Check type of expression item
         JM    Finis               Done if less than 0
         JP    Oprtr               Branch if an operator
         LR    1,WV                Move to GR1 for storing
Push     AHI   SP,-4               Push stack contents down once
         ST    1,Stack(SP)         Place new operand onto stack
Step     LA    EP,4(,EP)           Step pointer to next expression item
         J     Fetch               And get next item
*    Set up for operation on top two stack elements
Oprtr    L     1,Stack+4(SP)       Get next-to-top stack item
         L     2,Stack(SP)         Top stack element in GR2
         LA    SP,8(,SP)           Pop both elements off the stack
         LTR   WV,WV               Now check the operator code
         JZ    AddTwo              If zero, add operator
         MR    0,2                 Otherwise, multiply
         J     Push                And push the result back on stack
AddTwo   AR    1,2                 Add the operands
         J     Push                Go push sum onto stack
Finis    PrintOut 1,*              Print result and stop
         DS    (StkSize)F          Space for stack
Stack    DS    0F
Expressn DC    H'0,2,0,3,0,4,4,1,4,0,-1'    Expression 234*+
```

40.6.7. Suppose the top of the two stacks are represented by values Stk1T and Stk2T; the bottom of Stk1 is A(1) and the bottom of Stk2 is A(N). Then we can represent the four functions with this pseudo-code:

```
Push1(X): Stk1T = Stk1T+1;
          if Stk1T = Stk2T then Stk1 Overflow
          A(Stk1T) = X

Push2(X): Stk2T = Stk2T-1;
          if Stk2T = Stk1T then Stk2 Overflow
          A(Stk2T) = X

Pop1(X):  if Stk1T < 1 then Stk1 Underflow
          X = A(Stk1T)
          Stk1T = Stk1T-1;
```

```
Pop2(X):  if Stk2T > N then Stk2 Underflow
          X = A(Stk2T)
          Stk2T = Stk2T+1;
```

You can now write the appropriate instructions without difficulty.

Section 40.7

40.7.1. The **LstCount** field should be zero, and the **LstHLink** and **LstTLink** fields should both contain the address of **LstAnchr**.

40.7.5. The list is empty when both the forward and backward link addresses are the address of the list anchor.

40.7.10. We'll choose arbitrary values for the symbols **DataLen** and **NLItms** to show how the array can be allocated.

```
NLItms   Equ   20                 Assume a list of 20 elements
DataLen  Equ   40                 Assume data field length = 40
*
ArrStack DS    F,XL(DataLen)      Space for first list element
ElemLen  Equ   *-ArrStack         Length of each list element
         DS    (NLItms-1)XL(ElemLen)   Space for remaining elements
```

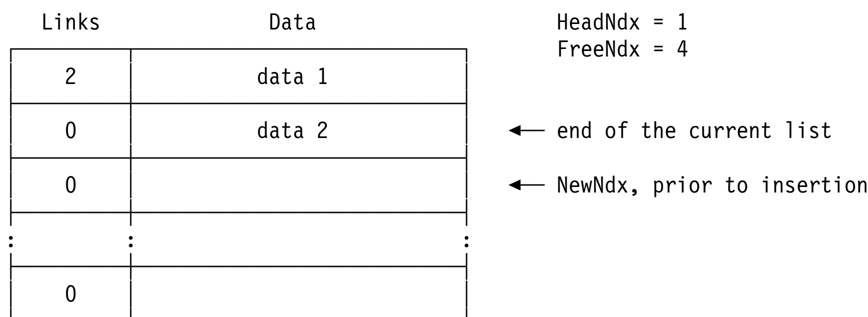
40.7.11. These instructions will initialize the list:

```
         LHI   0,NLItms-1         One less than all list elements
         LHI   2,2                Initial link value
         LA    1,ArrStack         Point to first list element
InitLoop ST    2,0(,1)            Store a successor index
         AHI   2,1                Increment the index
         AHI   1,ElemLen          Add the length of each list element
         JCT   0,InitLoop         Count down until last element
         ST    0,0(,1)            Last element has a null link
         ST    0,HeadNdx          Initialize head of working list
         LA    0,1                Index of first free-list element
         ST    0,FreeNdx          Initialize index of free-list head
```

40.7.12. The steps would be like these:

1. If the FreeNdx is zero, there are no more free elements; branch to **NoneLeft**
2. NewNdx ← FreeNdx [The index of the “New” element is taken from the head of the free list]
3. FreeNdx ← Link(FreeNdx) [The new head of the free list is the previous second element on the free list]
4. Link(NewNdx) ← 0 [Assume the new element might be the end of the list]

40.7.13. We have already added elements with indexes 1 and 2 to the list, and the first element of the free list has index 3. After taking this free element from the list, the list would look like this:



Having acquired the New element, you can use steps like these to add it to the head of the list:

1. Data(NewNdx) ← NewData [Put the new data in the new element]
2. Link(NewNdx) ← HeadNdx [The link in the new element points to the previous head element]
3. HeadNdx ← NewNdx [Make the new element the head of the list]

After adding the new element to the head of the list, the list would look like this:

Links	Data
2	data 1
0	data 1
1	New Data
:	:
0	

HeadNdx = 3
FreeNdx = 4

40.7.15. As written each element in the FSL links into the *data* field of the next element. If the second DC statement is corrected to read

```
         DC    (Elem_Len/4)A(0),(NLstItms-1)A(*-Elem_Len,0,0,0,0,0)
```

there will be no difference in the way the two lists are used.

Section 40.8

40.8.1. Only two links need to be updated:

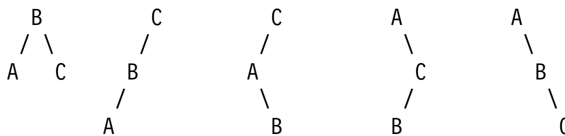
```
Q1       Using Q_El,5             Address of Q1
Q2       Using Q_El,9             Address of Q2
Q3       Using Q-El,2             Address of Q3
         MVC   Q1.RLink,Q2.RLink  Forward chain Q1 to Q3
         MVC   Q3.LLink,Q2.LLink  Backward chain Q3 to Q1
```

Section 40.9

40.9.1. It will make no difference, so long as elements are entered into the tree using the same left-right convention as when they are retrieved.

40.9.2. There are four other 3-node binary trees, for a total of five. Let the nodes be indicated by "*" characters:

40.9.3. These "inorder" configurations should help you understand how the order of traversal affects the trees:



You'll enjoy sketching these five trees for preorder and postorder traversal.

Section 40.10

40.10.1. You might use shift instructions this way:

```
HashEnts Equ   73                 Number of hash table entries
         L     0,DataItem         Load data item into GR0
         SRDL  0,16               Last 2 bytes of data item in GR0
         SRL   1,16               First 2 bytes of data item in GR1
         XR    1,0                XOR the halves into GR1
         XR    0,0                Clear GR0 for a division
         D     0,=A(HashEnts)     Divide by hash table size
```

Not a significant savings, but one less memory reference may help.

40.10.2. If the high-order bit of one of the 2-byte fields is 1 (but not in both fields), a different but still valid hash index will be generated.

Programming Problem 40.1.

The main feature of this solution is that the size of the graph is defined by the two EQU statements at the beginning. All other quantities that depend on the length and width of the graph are then defined in terms of those values.

```
         Title 'Solution to Problem 40.1'
P40_1    Start 0                 Graph-plotting problem
*
XLen     Equ   119               Number of X values
YLen     Equ   59                Number of Y values
*
         BASR  12,0
         Using *,12              Inform assembler
         PrintLin,Page,L'Page    Print title line
         LA    11,XLen           Initial X position at 'XLen'
XLoop    LR    10,11             Carry X value in GR10
         SH    10,MiddlX         Form true X value from index
         JNZ   NotYAxis          If X not zero, no Y axis yet
         LA    2,Graph+(XLen+1)/2 Position of Y-axis
         LA    3,YLen            Number of lines
*
YLoop    MVI   0(2),C'|'         Put Y-axis marker in line
         LA    2,XLen(0,2)       Increment by line length
         JCT   3,YLoop           Do YLen times
NotYAxis LR    1,10              Set up for parabola calculation
         MR    0,1               X squared
         AHI   1,25              Add roundoff factor
         D     0,F50             Divide by 50
         AHI   1,-25             Subtract 25 giving -y
         AH    1,HalfY           Add YLen/2 to get line index
         JM    NotStar           If negative, off top of page
         CH    1,HYLen           See how close to bottom
         JNL   NotStar           If bigger than YLen, off low end
         MH    1,HXLen           Otherwise multiply by line length
         LA    2,Graph-1(11)     Address of X position in top line
         AR    2,1               Add line index
         MVI   0(2),C'*'         Store asterisk in graph
NotStar  LCR   1,10              Set up for straight line
         AHI   1,-12             Subtract 12
         SRA   1,1               Divide by 2, giving -Y
         AH    1,HalfY           Add YLen/2 to get line index
         BM    NotX              If negative, off the top
         CH    1,HYLen           See if off the bottom
         JNL   NotX              Jump if so
         MH    1,HXLen           Mult line index by line length
         LA    2,Graph-1(11)     Construct X-position in top line
         AR    2,1               Add line factor to get array loc
         MVI   0(2),C'X'         Store 'X' in the array
NotX     JCT   11,XLoop          Count down on X position
         LA    4,YLen            Set up for printing Ylen lines
         LA    3,Graph           Establish initial line address mover
Mover    MVC   Line+1(XLen),0(3) Move line to print position
         PrintLin Line,L'Line    Print line with space at left
         LA    3,XLen(0,3)       Increment address
         JCT   4,Mover           Loop until all lines are done
         Printout *,Header=NO    Exit
*
Page     DC    C'1 Solution to Problem 40.1'     Heading line
         DC    CL(XLen+1)' '     Space for lines to be printed
F50      DC    F'50'
HXLen    DC    AL2(XLen)
HalfX    DC    AL2(XLen/2)
HYLen    DC    AL2(YLen)
HalfY    DC    AL2(YLen/2)
```

```
MiddlX   DC    AL2((XLen+1)/2)
Graph    DC    (YLen/2)CL(XLen)' '
         DC    (XLen)C'-'          X-Axis of minus signs
         DC    (YLen/2)CL(XLen)' '
         End   P40_1
```

Programming Problem 40.2.

```
         Title 'Solution to Problem 40.2'
*        As each character is read from the input record, it is
*        immediately converted to a numeric index which controls
*        all further operations. This master index selects the
*        operation, determines the stack increment, and chooses
*        the operands.
*
P40_2    START 0
StkSiz   Equ   20                  Size of stack
         BASR  12,0                Use register 12 for base
         Using *,12
         J     Begin               Branch around data definitions
Formula  DC    0CL82' ',C'0'       Define print line and carriage ctrl
Record   DC    CL80' ',C' '        Define record, 1 blank to force stop
Master   DC    256X'0'             Initialize table to all zeros
         ORG   Master+C'A'         Variable A
         DC    X'1'
         ORG   Master+C'B'         Variable B
         DC    X'2'
         ORG   Master+C'C'         Variable C
         DC    X'3'
         ORG   Master+C'0'         Constants 0,1
         DC    X'0405'
         ORG   Master+C'N'         NOT operation
         DC    X'6'
         ORG   Master+C'&&'        AND operation
         DC    X'7'
         ORG   Master+C'|'         OR operation
         DC    X'8'
         ORG   Master+C'X'         XOR operation
         DC    X'9'
         ORG   *                   Set LC to end of translate table
*
Stack    DC    XL(StkSiz)'0'         Stack
StkLim   DC    Y(StkSiz)             Size of stack, for checking
Data     DC    X'F0CCAA00FFFF'       Values for A,B,C,0,1,NOT
Incr     DC    H'0,1,1,1,1,1,0,-1,-1,-1' Stack ptr increments
IncrX    DC    AL1(0,2,4,6,8,10,12,14,16,18) Offsets into incr
Branch   DC    AL1(EndForm-Operate) Offset for all other chars
         DC    AL1(OprndA-Operate)  Offset for 'A'
         DC    AL1(OprndB-Operate)  Offset for 'B'
         DC    AL1(OprndC-Operate)  Offset for 'C'
         DC    AL1(Oprnd0-Operate)  Offset for '0'
         DC    AL1(Oprnd1-Operate)  Offset for '1'
         DC    AL1(NOT-Operate)     Offset for 'N'
         DC    AL1(AND-Operate)     Offset for '&'
         DC    AL1(OR-Operate)      Offset for '|'
         DC    AL1(XOR-Operate)     Offset for 'X'
M0       DC    C' Result is always False'
M1       DC    C' Result is always True'
MM       DC    C' Result is indeterminate'
Quit     DC    C'0End of tests'
StkMsg   DC    C' *** Stack Overflow'
SynMsg   DC    C' *** Stack holds more than 1 item at end'
MsgList  DS    0F                    Start of message list
M0P      DC    AL1(L'M0),AL3(M0)     Result all zero
M1P      DC    AL1(L'M1),AL3(M1)     Result all ones
```

```
MMP      DC    AL1(L'MM),AL3(MM)    Result mixed bits
*
*        Register usage -- (all are set to 0 when record is read)
*        R0 = Result of operation
*        R1 = Master index and branch index
*        R2 = Stack index increment table index
*        R3 = Stack pointer
*        R4 = Old stack top value
*        R5 = Data value for variables and constants
*        R6 = Input record scan pointer and message index
*
Begin    ReadCard Record,Stop     Read a record, stop if no more
         PrintLin Formula,L'Formula
         LM    0,6,=7F'0'         Initialize all registers to 0
Fetch    IC    1,Record(6)        Get a character from the record
         IC    1,Master(1)        Replace it by its master index
         IC    2,IncrX(1)         Get (stack index increment) index
         AH    3,Incr(2)          Add increment to stack index
         BNP   StkErr             Stack underflow if not positive
         CH    3,StkLim           Compare to stack size
         BH    StkErr             Stack overflow if exceeds StkSiz
         IC    0,Stack-1(3)       Get new stack top
         IC    4,Stack(3)         Get old stack top
         IC    5,Data-1(1)        Get operand, if any
         IC    1,Branch(1)        Now set up branch index
******   PrintOut 0,1,2,3,4,5,6,Stack    for debugging
         B     Operate(1)         And go do the operation
Operate  Equ   *                  Start of operation list
OprndA   Equ   *                  Character is 'a'
OprndB   Equ   *                  Character is 'b'
OprndC   Equ   *                  Character is 'c'
Oprnd0   Equ   *                  Character is '0'
Oprnd1   Equ   *                  Character is '1'
         LR    0,5                Move data to result register
         B     EndOps             And go complete the operation
AND      NR    0,4                Perform 'AND' operation
         B     EndOps             Go store the result
OR       OR    0,4                Perform 'OR' operation
         B     EndOps
NOT      XR    0,5                Perform 'NOT' operation
         B     EndOps
XOR      XR    0,4                Perform 'XOR' operation
EndOps   STC   0,Stack-1(3)       Store result back on stack
******   PrintOut 0,1,2,3,4,5,6,Stack  for debugging
         LA    6,1(0,6)           Increment record scan pointer
         B     Fetch              And go get next character
StkErr   PrintLin StkMsg,L'StkMsg Indicate stack error
         B     Begin              And start with a new record
SynErr   PrintLin SynMsg,L'SynMsg Bad formula
         B     Begin              Read next formula
EndForm  CHI   3,1                Check final stack index
         BNE   SynErr             Syntax error, stack holds too much
         LA    6,M0P-MsgList      Set for result being all zeros
         TM    Stack,X'FF'        Test all bits on first stack entry
         BZ    PRINT
         LA    6,MMP-MsgList      Assume now the bits are mixed
         BM    PRINT              Branch if that's true
         LA    6,M1P-MsgList      Otherwise they're all ones
PRINT    L     1,MsgList(6)       Get pointer to message
         IC    2,MsgList(6)       And message length
         PrintLin 0(1),0(2)       Print result message
         B     Begin              And start all over again
Stop     PrintLin Quit,L'Quit     That's all, give up
         BR    14                 Return to supervisor
         End   P40_2
```

Programming Problem 40.6.

This solution uses indentation to show the nesting of the three loops. The size of the matrices is defined by the EQU statement defining N.

```
P40_6    CSect ,
         Using *,15
N        Equ   5
*                                Begin Outer loop
         LHI   1,N               Initialize I in GR1
*                                  Begin Middle loop
LOuter     LHI   2,N               Initialize J in GR2
LMiddle      XR    0,0               Accumulated sum in GR0
             LHI   3,N               Initialize K in GR3
*                                      Inner loop: calculate C(I,J)
LInner         LR    4,1               I
               BCTR  4,0               I-1
               MHI   4,N               (I-1)*N
               ALR   4,3               (I-1)*N+K
               BCTR  4,0               (I-1)*N+K-1
               SLL   4,2               4*((I-1)*N+K-1)
               L     7,A(4)            Get A(I,K)
               LR    5,3               K
               BCTR  5,0               K-1
               MHI   5,N               (K-1)*N
               ALR   5,2               (K-1)*N+J
               BCTR  5,0               (K-1)*N+J-1
               SLL   5,2               4*((K-1)*N+J-1)
               M     6,B(5)            A(I,K) * B (K,J)
               AR    0,7               Accumulate sum
               JCT   3,LInner          Iterate on K
*                                      End of Inner loop
             LR    4,1               I
             BCTR  4,0               I-1
             MHI   4,N               (I-1)*N
             ALR   4,2               (I-1)*N+J
             BCTR  4,0               (I-1)*N+J-1
             SLL   4,2               4*(I*N+J)
             ST    0,C(4)            Store C(I,J)
             JCT   2,LMiddle         Iterate on J
*                                    End of Middle loop
           JCT   1,LOuter          Iterate on I
*                                End of Outer loop
         DumpOut C,C+4*(N*N)     Display result in hex format
         PrintOut *,Header=NO
A        DC    (N)F'1,2,3,4,5'
B        DC    (N)F'5,4,3,2,1'
C        DS    (N*N)F
         End   P40_6
```

Programming Problem 40.8.

```
*        Solution to 'Bordered-Square' Programming Problem 40.8.
* This program is heavily parameterized, so that you need only to
* change the value of N, the length of each row of the square.
* If N exceeds 58 or is less than 6, various assembly errors occur.
*
         Print NoGen
P40_8    Start 0                 Print a square
         Using *,15
*
N        Equ   12                Length of one edge
SqSize   Equ   (N/2)*2           Ensure square size is an even value
NZeros   Equ   SqSize-2          Number of inner zeros to print
```

```
Lines    Equ   60                Assume 60 lines per page
TopRow   Equ   (Lines-SqSize)/2-1 Line number of top of square
LineLen  Equ   120               Length of the line
LineCntr Equ   LineLen/2         Center position on a line
OuterCh  Equ   C'1'              Character for outer border
InnerCh  Equ   C'0'              Character for inner border
LeftEdge Equ   LineCntr-(SqSize/2)     Left edge of printed square
*
         PrintLin TopLine,1      Start a new page
         LA    1,TopRow          Calculate number of lines to skip
         SR    0,0               Clear high-order word
         D     0,=F'3'           Divide by 3
         LTR   1,1               Any lines to triple-space?
         JZ    NoSkip3
Skip3    PrintLin CCChars,1      Triple space
         JCT   1,Skip3           Continue triple-spacing as needed
NoSkip3  LTR   0,0               Any lines to single/double space?
         JZ    NoSkips           If not, done with skipping lines
         LA    1,CCChars         Point to CCChars
         AR    1,0               Index to the needed CC character
         PrintLin 0(1),1         Do the final skip
NoSkips  MVI   Square1+LeftEdge,OuterCh     Start outermost row
         MVC   Square1+LeftEdge+1(SqSize-1),Square1+LeftEdge fill row
         MVC   Square2,Square1   Copy the row
         MVI   Square2+LeftEdge+1,InnerCh   Start second row
         MVC   Square2+LeftEdge+2(NZeros-1),Square2+LeftEdge+1 fill row
         MVC   Square3,Square2   Copy the second row
         MVI   Square3+LeftEdge+2,C' ' Blank first interior character
         MVC   Square3+LeftEdge+3(SqSize-5),Square3+LeftEdge+2 fill
*
         MVC   OutLine,Square1   Copy top row
         PrintLin OutBuf,L'OutBuf Print the line
         MVC   OutLine,Square2   Copy second row
         PrintLin OutBuf,L'OutBuf Print the line
         MVC   OutLine,Square3   Copy next row
         LA    1,SqSize-4        Number of innermost rows
PrtInner PrintLin OutBuf,L'OutBuf Print the line
         JCT   1,PrtInner
         MVC   OutLine,Square2   Copy second row
         PrintLin OutBuf,L'OutBuf Print the line
         MVC   OutLine,Square1   Copy top row
         PrintLin OutBuf,L'OutBuf Print the line
         PrintOut *,Header=NO    Terminate the program
*
OutBuf   DC    CL(LineLen+1)' '  Initialize output buffer
OutLine  Equ   OutBuf+1,LineLen  Define buffer position of output
Square1  DC    CL(LineLen)' '    Top and bottom rows of square
Square2  DC    CL(LineLen)' '    Second and N-1th rows of square
Square3  DC    CL(LineLen)' '    Remaining rows of square
TopLine  DC    C'1'
CCChars  DC    C'- 0'            CC characters for 1/2 spaces
         End   P40_8
```

Section 41 Solutions

Section 41.2

41.2.2. The first argument R is positional; the other two arguments A= and LV= are keyword.

Section 41.3

41.3.2. Because the expansion of the OPEN macro refers to it, and you don't want the assembly to fail due to an undefined symbol.

Section 41.4

41.4.1. The address X'BAD' is odd, so branching to it will cause a specification exception. The address X'D1E' is even and will cause a branch into the low area of memory, with unpredictable results.

41.4.2. The address C'BAD' is even, so it will cause a branch into the low area of memory, with unpredictable results. The address C'D1E' is odd and will cause a specification exception.

41.4.3. The displacement in the LA instruction at offset X'000000' (statement 6) is X'04F', but it should be X'02A'.+

41.4.4. The target of the branch instruction is its third halfword, X'0002'.

Section 41.5

41.5.1. If executed in 24-bit addressing mode, the BAL instruction places the Instruction Length Code B'10' (BAL is 4 bytes long) in the two high-order bits of the R_1 register. In 31-bit mode, the high-order bit of GR R_1 is set from the Basic Addressing Mode bit of the PSW, which is 1 for 31-bit mode.

Section 41.7

41.7.1. Some possible but unverified instruction sequences are:

```
(01)      DC   H'0'          Invalid operation
(02)      LPSW 0,8(0,0)      Privileged operation
(03)      EX   0,*           Execute
(04)      MVI  0,0           Protection
(05)      LGHI 1,-1          Addressing
          IC   1,0(,1)
(06)      DC   X'1D11'       DR 1,1: Specification
(07)      AP   *,*           Decimal Data
(08)      LA   1,1           Fixed-point overflow
          SLA  1,33
(09)      SRDL 0,63          Fixed-point divide
          DR   0,0
(0A)      AP   A,A           Decimal overflow
        A DC   P'6'
(0B)      DP   A,=P'0'       Decimal divide
(0C)      LE   0,=E'1E72'    Exponent overflow
          MER  0,0
(0D)      LE   0,=E'1E-72'   Exponent underflow
          MER  0,0
(0E)      LE   0,=X'41000001' Significance
          SU   0,=X'41000001'
(0F)      LE   0,=E'1'       Floating-point divide
          D    0,=E'0'
```

+ I revised an earlier version of the figure and decided to leave the old displacement in place for you. Now: What was the *original* value of the ABEND code?

Programming Problem 41.2.

This program shows one way to read 80-byte records and display the results. It uses three conditional-assembly SETA statements to set values of the input and output record lengths and the program name.

```
*********************************************************************
* Read and list 80-character fixed-length records, with a sequence   *
* number preceding the record image in the listing.                  *
*********************************************************************
&OutLen  SetA  121                Length of print line
&InLen   SetA  80                 Length of input records
&Name    SetC  'Prob41_2'         Name of program
&Name    CSect ,
&Name    AMode 24                 Set required addressing mode
&Name    RMode 24                 Set required residence mode
         Print NoGen              DCB, DCBD expansions are VERY long!
*
         SAVE  (14,12)            Save caller's registers
         LR    12,15              Copy entry address
         Using &Name,12           Establish base address
         LR    11,13              Copy caller's save area address
         LA    13,SaveArea        Point to local save area
         ST    11,SaveArea+4      Store backward link in our area
         ST    13,8(,11)          Store forward link in caller's area
         OPEN  (OutDCB,(OUTPUT))  Open output DCB
Out      Using IHADCB,OutDCB      Map the output DCB
         TM    Out.DCBOFLGS,DCBOFOPN   Did the output DCB open OK?
         JO    OpenIn             Yes, open the input DCB
         ABEND 1,DUMP             No sense trying to go further
OpenIn   OPEN  (InDCB,(INPUT))    Open input DCB
In       Using IHADCB,InDCB       Map the input DCB
         TM    In.DCBOFLGS,DCBOFOPN    Did the input DCB open OK?
         Drop  Out,In             No more DCBD mappings needed
         JO    Proceed            Yes, continue
         PUT   OutDCB,OpenMsg     Tell the user the open failed
         CLOSE OutDCB             Close the output DCB
         J     Finish             Say Farewell
Proceed  XR    2,2                Set record count to zero
GetaRec  GET   InDCB,InRec        Read a record
         AHI   2,1                Increment record count
         CVD   2,DTemp            Convert to packed decimal.
         MVC   RecNum,NumPat      Move edit pattern to line
         ED    RecNum,DTemp+5     Edit 5 digits
         PUT   OutDCB,OutLine     Print the output line
         J     GetaRec            Go read another record
EOF      CLOSE (InDCB,,OutDCB)    Close both DCBs
         FREEPOOL InDCB           Free input buffers
Finish   FREEPOOL OutDCB          Free output buffers
         L     13,SaveArea+4      Point to caller's save area
         LM    14,12,12(13)       Restore caller's registers
         XR    15,15              Set return code to zero
         BR    14
*
NumPat   DC    X'402020202120'    Up to 10**5-1 records
*
OutLine  DS    0CL(&OutLen)       Start of output line
RecNum   DS    CL(L'NumPat)       Record number + carriage control
         DC    CL3' '             Spaces
InRec    DS    CL(&InLen)         Input record to be listed
         DC    CL(&OutLen-(*-OutLine))' '   Blanks for rest of line
OpenMsg  DC    CL(&OutLen)'1Unable to open input DCB' We can't print
*
InDCB    DCB   DDNAME=INPUT,LRECL=&InLen,RECFM=FB,DSORG=PS,MACRF=GM,   X
```

```
               EODAD=EOF,BLKSIZE=&InLen*10
OutDCB   DCB   DDNAME=OUTPUT,LRECL=&OutLen,RECFM=FBA,DSORG=PS,MACRF=PM,X
               BLKSIZE=&OutLen*10
*
SaveArea DC    9D'0'              Local save area
DTemp    DS    D                  Work area for CVD
         DCBD  DSORG=PS,DEVD=DA   Generate IHADCB DSect
         End   &Name.
```

Using the data in Programming Problem 24.14 on page 400, the output of this program was:

```
     1   Four score and seven years ago our fathers brought forth on this continent a ne
     2   w nation, conceived in liberty, and dedicated to the proposition that all men a
     3   re created equal. Now we are engaged in a great civil war, testing whether that
     4    nation, or any nation, so conceived and so dedicated, can long endure. We are
     5   met on a great battle-field of that war. We have come to dedicate a portion of
     6   that field, as a final resting place for those who here gave their lives that t
     7   hat nation might live. It is altogether fitting and proper that we should do th
     8   is.
     9
    10   But, in a larger sense, we can not dedicate, we can not consecrate, we can not
    11   hallow this ground. The brave men, living and dead, who struggled here, have co
    12   nsecrated it, far above our poor power to add or detract. The world will little
    13    note, nor long remember what we say here, but it can never forget what they di
    14   d here. It is for us the living, rather, to be dedicated here to the unfinished
    15    work which they who fought here have thus far so nobly advanced. It is rather
    16   for us to be here dedicated to the great task remaining before us -- that from
    17   these honored dead we take increased devotion to that cause for which they gave
    18    the last full measure of devotion -- that we here highly resolve that these de
    19   ad shall not have died in vain -- that this nation, under God, shall have a new
    20    birth of freedom -- and that government of the people, by the people, for the
    21   people, shall not perish from the earth.
```

Section 42 Solutions

Section 42.1

42.1.1. Try `XC  *,=X'01'`. The XC instruction changes alternately into OC, XC, OC, ...

Similarly, `XI *,1` alternates between OI, XI, OI, ...

What property of these two pairs of opcodes makes this possible?

You will of course remember that modifying instructions is a *very* poor practice that slows execution considerably, and makes the program non-reenterable.+

Section 42.2

42.2.1. By convention, GR15 must contain the entry point address when entering a subprogram. The entry point address is used as a base address for any instructions (particularly in the macro expansions) that must be resolved using base-displacement addressing.

42.2.3. You can see that the GETMAIN macro generates a BAL instruction that requires base-displacement resolution. The FREEMAIN macro generates a LA instruction with explicit operands. All the other instructions either address the WA DSect using GR13 as a base register, or use relative addressing within the RSect. Thus, the earliest position for the DEOP 3 statement is immediately following the GETMAIN instruction.

Programming Problem 42.1

This factorial program was written to manage its own internal stack and call itself recursively.

```
FactA    CSect ,
*******************************************************************
* Evaluate N! recursively. The largest valid N is 12.             *
*******************************************************************
         Using *,15
         ST    14,Temp            Save return address
         L     1,0(,1)            Get parameter address
         L     1,0(,1)            Get N
         ST    1,Temp+4           Save current value of N
         ST    2,Temp+8           Save current value of GR2
         L     2,StackPtr         Get stack pointer
         AHI   2,12               Increment stack pointer
         MVC   0(12,2),Temp       Save return address, N, GR2
         ST    2,StackPtr         Save new stack pointer
         CHI   1,1                Compare N to 1
         JH    Recur              If greater, call again
         LA    0,1                1 Factorial = 1
         J     Exit               Return
*******************************************************************
Recur    BCTR  1,0                N-1                             *
         ST    1,Data             Save for recursive call         *
         LA    1,DataPtr          Point to argument address       *
         BASR  14,15              Call ourselves recursively      *
*******************************************************************
         L     2,StackPtr         Retrieve stack pointer
         L     1,4(,2)            Get value of N from call
         MR    0,0                Form N*Fact(N-1)
         LR    0,1                Form return value
*
Exit     L     2,StackPtr         Retrieve stack pointer
         L     14,0(,2)           Get proper return address
         AHI   2,-12              Decrement stack pointer
         ST    2,StackPtr         Save updated stack top pointer
         L     2,12+8(,2)         Restore caller's GR2
```

+ Doing things like that could also endanger your job security.

```
         BR    14                 Return to caller
*
Temp     DS    2F                 Word 1 = return addr, Word 2 = N
Data     DC    F'0'               Value of N for next call (it's N-1)
DataPtr  DC    A(Data)            Address of data item for call
StackPtr DC    A(Stack-8)         Initial stack pointer
Stack    DC    (13*3)F'0'         Fact(13) is too big, anyway
         End
```

This is a calling program:

```
CallFac  CSect ,
         Using *,12               Establish addressability
         LR    12,15              Copy base register
         LA    5,12               Max N that fits in a fullword = 12!
Test     ST    5,Value            Store N for call
         ConvertO 5,LineN         Convert N to characters
         LA    1,AP               Point to argument list
         L     15,AF              Address of factorial routine
         BASR  14,15              Call it
         ConvertO 0,LineFac       Convert result to characters
         PrintLin Line,Linelen    Print the result
         JCT   5,Test             Reduce N by 1 and repeat
         Printout *,Header=No     Terminate caller
AF       DC    V(FactA)           Address of factorial routine
AP       DC    A(Value)           Address of N
Value    DS    F                  Argument N

LinePos  DS    0C                 Position of the output line
LineN    DC    CL12' '            Carriage control, Value of N
         DC    C' factorial = '
LineFac  DC    Cl12' '            Value of N factorial
LineLen  Equ   (*-Line)           Line Length
         Org   LinePos            Step back to line position
Line     DS    0CL(LineLen)       Define symbolic output line length
         Drop  12
         End
```

The output from this program is:

```
          12 factorial =    479001600
          11 factorial =     39916800
          10 factorial =      3628800
           9 factorial =       362880
           8 factorial =        40320
           7 factorial =         5040
           6 factorial =          720
           5 factorial =          120
           4 factorial =           24
           3 factorial =            6
           2 factorial =            2
           1 factorial =            1
```

Programming Problem 42.3

Mine created a storage protection error for a very remote address.

Index

Special Characters

' apostrophe
in attribute reference 104
in C-type constants 150
in self-defining terms 86
− minus sign
in expressions 97
negation operator 13
unary 97
_ underscore
in symbols 89
/ slash, solidus
division operator 13
in expressions 97
( ) parentheses
address constant delimiters 147
in expressions 97
× product sign
multiplication operation 13
@ at sign
in symbols 89
non-invariant character 432
• bullet character
representation of blank character 13
representation of space character 13
$ dollar sign
$$GENIO macro instruction
definition 1028
description 1023
in symbols 89
non-invariant character 432
* asterisk
in expressions 97
location counter reference 97
multiplication operator 97
*PROCESS statement 1051
& ampersand
in C-type constants 150
in self-defining terms 86
hash, sharp, (US) pound sign
in symbols 89
non-invariant character 432
+ plus sign
addition operator 13
in expressions 97
unary 97
= equal sign
literal indicator 154
| | vertical bars
absolute value operator 13
÷ quotient sign
division operation 13

A

A machine instruction 216
A-type address constant
definition 866
ABEND
definition 981
ABEND macro instruction 956, 977
abnormal termination 972
ABEND macro instruction 956, 972
any kind 976
definition 981
memory dump 956
ABS
absolute value operator 13
absolute implied address 131
absolute USING location 130
access exception 57
access register 48
ACONTROL assembler instruction 1051
definition 1041
AD machine instruction 606
ADATA assembler option 89
ADB machine instruction 661
ADBR machine instruction 661
adcon
See also address constant
definition 158, 1041
addend
definition 238, 1041
addition
binary floating-point 661
decimal floating-point 703
fixed-point binary 216
fixed-point binary integers 31
floating-point 578
hexadecimal floating-point 606, 612
complement addition 614
process 612
immediate 321
logical 224
packed decimal 485
packed decimal operand order dependence 486
packed decimal process 492
register-immediate 321
with carry 228
address 92
assembly time 1041
definition 1041
execution time 92, 1041
location 1041
vs. location 92
address constant
definition 158, 866, 1041
delimited by parentheses 147
expression as nominal value 147
S-type 149
type A 147
type AD 157
type V 820
type Y 149
address generation 302
address generation *(continued)*
base-displacement addressing 62
execution time 116
relative-immediate 302, 305
signed 20-bit displacement 302
unsigned 12-bit displacement 62, 302
address resolution
addressable 126
addressing halfword 1041
base-displacement form 126
definition 1041
displacement 123
DSECT 874
explicit address 119
highest-numbered register 128
implied address 120, 126
implied addresses 1041
multiple 127
qualified symbol 882
relocation attribute 125
rules 132, 303
signed 20-bit displacement 303
smallest displacement 128
unsigned 12-bit displacement 62
valid displacement 126
address table 917
definition 944, 1041
address translation 67
DAT 67
definition 68, 1041
virtual address 67
address vs. location
execution vs. assembly time 92
addressability 66, 132
definition 68, 133, 1041
error 129, 1041
definition 133
addressable
assembly time 1042
base-displacement resolution 1042
definition 1042
execution time 1042
resolution 1042
addressing
addressability 66
base address 62
base register 62
base-displacement addressing 62, 1043
base-displacement format 1043
Effective Address 62
Effective Address Register 62
in large programs 790
index 63
indexed Effective Address 64
internal subroutine 798
mode 307
not addressable 66
relative-immediate 305

addressing *(continued)*
with address constants 793
without local addressability 797
addressing halfword 53, 62
base register specification digit 62
definition 69, 1042
signed 20-bit displacement 302
unsigned 12-bit displacement 62
addressing mode 307
24-bit 307
31-bit 307
64-bit 307
address generation 307
AMODE 827
AMODE assembler instruction 827
definition 1042
entry symbols 850
ADR machine instruction 606
ADTR machine instruction 701
ADTRA machine instruction 701
AE machine instruction 606
AEB machine instruction 661
AEBR machine instruction 661
AER machine instruction 606
AFI machine instruction 321
AG machine instruction 216
AGF machine instruction 230
AGFI machine instruction 321
AGFR machine instruction 230
AGHI machine instruction 321
AGR machine instruction 216
AH machine instruction 216
AHI machine instruction 321
AL machine instruction 224
ALC machine instruction 228
ALCG machine instruction 228
ALCGR machine instruction 228
ALCR machine instruction 228
ALFI machine instruction 321
ALG machine instruction 224
ALGF machine instruction 230
ALGFI machine instruction 321
ALGFR machine instruction 230
algorithm
definition 14, 1042
ALGR machine instruction 224
ALIAS assembler instruction
external symbol renaming 1046
alignment 44
automatic 140
by CNOP 208
by NOALIGN option 139
by ORG 167
by SECTALGN option 139
CNOP and SECTALGN 809
DC and SECTALGN 809
doubleword 44
DS and SECTALGN 809
fullword 44
halfword 44
implied length 140
instructions 50
ORG and SECTALGN 809
quadword 44
alignment *(continued)*
SECTALGN option 809
word 44
ALR machine instruction 224
AMODE
definition 866, 1042
AMODE assembler instruction
See addressing mode
ampersand
in C-type constants 150
in self-defining terms 86
anchor
base location 1042
base register 1042
definition 1042
AND operation 289, 298, 1042
definition 298, 1042
OR operation
definition 298
register-immediate 323
register-register 290
storage-immediate 354
storage-storage 376
AP machine instruction 497, 501
apostrophe
in C-type constants 150
in self-defining terms 86
AR machine instruction 216
architecture
definition 5, 14, 1042
argument 765
definition 788, 1042
argument address list 766
argument passing 759
arithmetic
binary addition 31
binary division 274
binary floating-point special values 647
binary multiplication 265
binary subtraction 32
decimal floating-point 701
double-length shift 252
hexadecimal floating-point 591
packed decimal 484
packed decimal addition 492
packed decimal arithmetic rules 484
packed decimal subtraction 493
real vs. realistic floating-point arithmetic 745
scaled packed decimal 522
general rules 522
shifts 252
single-length shift 252
arithmetic division
definition 284, 1042
process 280
arithmetic multiplication
definition 284, 1042
process 272
arithmetic representation 28
definition 39, 1042
arithmetic shift
definition 261, 1042
arithmetic shift *(continued)*
process 244
array 908
address table 917
as a list of items 908
as a table 914
binary search 920
column order 910
definition 944, 1042
general subscripts 913
multi-dimensional 913
non-homogeneous 914
one-dimensional 908
row order 910
search 919
subscripting function 911
two-dimensional 910
ASCII
C-type constant 151
CA-type constant 157
numeric character 462, 478, 479
definition 483, 1042
pack 478
representation 432
table 1013
translation 433
unpack 479
Assembler 4
definition 14, 82, 1042
input record fixed length 75
assembler instruction
CSECT 80
END 80
ORG
extended syntax 168
START 80
assembler instruction statement 74
ACONTROL 1041
AMODE 827
CNOP 208
COM 803
CSECT 803
CSECT and LOCTR 811
DC 137
DROP 128, 883, 890, 905
DS 159
DSECT 804, 874
DXD 804
ENTRY 821
EQU 93, 162
EXTRN 818
difference from WXTRN 820
LOCTR 810
LOCTR and CSECT 811
LTORG 156
ORG 167
RMODE 827
RSECT 803, 984
START 803
TITLE 78
USING 120, 126, 882, 885, 891
WXTRN 818
difference from EXTRN 820
Assembler Language 4
definition 14, 1042

assembler option 133
  *PROCESS statement 1051
  ACONTROL assembler instruction 1051
  ADATA 89
  CODEPAGE 439
  COMPAT(TRANSDT) 88, 433
  DBCS 436
  GOFF 809, 849, 854
  NOALIGN 139
  NOTHREAD 810, 1252
  RENT 984
  SECTALGN 139, 156, 809
  TEST 89
  TRANSLATE 88, 158, 433
  USING(MAP) 133
assembly
  External Symbol Dictionary 93
  input records 75
  invocation 72
  location counter vs. instruction address 92
  object code 73
  process 72
    pass 1 123
    pass 2 125
  separate 802
  statements 73
assembly time 74
  definition 82, 1042
  location 92
assisted linkage 783, 847
  V-type address constant 820
attribute
  assembler 166
  definition 1042
  integer 90
  length 90, 143
  program 138, 166
  reference 90
  relocation 90, 125
  scale 90, 467
  type 90
  value 90, 125
attribute reference
  See also attribute reference
  definition 1042
  integer 96
  length 96, 155
  scale 96
  to literals 155
augend
  definition 238, 1042
AXBR machine instruction 661
AXR machine instruction 606
AXTR machine instruction 701
AXTRA machine instruction 701

B

B extended mnemonic 210
B-tree 940
  definition 944, 1042
BAS machine instruction 758
base address 117, 126
base address *(continued)*
  definition 69, 1042, 1043
  displacement 1042
  effective address 1043
  first operand of USING statement 1043
  general purpose register 1042
base digit 62
  See also base register specification digit
  definition 1043
base location
  base-displacement address resolution 1043
  definition 1043
  dependent USING statement 1043
  displacement 1043
  in USING assembler instruction 126
  ordinary USING statement 1043
base register 62, 120
  definition 69, 1043
base register specification digit 62, 109
  definition 69, 1043
base register zero 131
base-displacement addressing 62
  definition 1043
base_location 120
  absolute 130
  definition 133
base_register
  definition 133
BASR machine instruction 116, 758
BASSM machine instruction 858
BC machine instruction 204, 205
BCD representation 429
BCR machine instruction 204, 205
BCT machine instruction 334
BCTG machine instruction 334
BCTGR machine instruction 334
BCTR machine instruction 334
BE extended mnemonic 211
BEAR 212
begin column 75
BER extended mnemonic 211
BH extended mnemonic 211
BHR extended mnemonic 211
bias 575, 584
  definition 585, 1043
  exponent 584
  rounding 575
biased rounding
  definition 530, 1043
Big-Endian 453
  definition 1043
binary
  floating-point 638
  integer addition 31
  integer subtraction 32
  overflow detection recipe 31
  shift 242
  subtraction recipe 35
  two's complementation recipe 27
binary digit
  See bit
binary floating-point
  addition 661
  characteristic 639
    reserved values 640
  compare and signal 663
  comparison 662
  constant 643, 644, 645
    length modifier 645
    modifiers 644, 645
    rounding-mode suffix 643
    special values 643
    type DB 642
    type EB 642
    type LB 642
  convert to decimal floating-point and hexadecimal floating-point 743
  converting from integer 666
  converting to integer 666
  data classes
    denormalized numbers 639
    infinity 639
    normal numbers 639
    quiet NaN 639
    signaling NaN 639
    zero 639
  Data Exception Code 650
  data formats 638
  decimal exponent 644
  definition 1043
  divide to integer 669
  division 659
  division by zero 649
  exception handling 650
  exponent field width 639
  exponent modifier 644
  exponent overflow 649
  exponent underflow 649
  floating-point integer 668
  inexact result 649
  infinity 641
  invalid operation 649
  lengthening instructions 664
  multiplication 657
  multiply and add/subtract 672
  NaN (Not a Number) 639, 640
    payload 644
    quiet 641
    signaling 641
  overview 646
    arithmetic with special values 647
    denormalized numbers 647
    exceptions 649
    rounding modes 646
  remainder 668
  representation 639
  representation range 641
  rounding instructions 664
  set rounding mode 651
  signed zero 748
  significand 639
  special values 639, 640
    denormalized numbers 640

binary floating-point *(continued)*
special values *(continued)*
infinity 640
NaN 640
square root 671
subtraction 661
test data class 654
binary search 920
definition 944, 1043
binary self-defining term
See self-defining term
binary tree 937
definition 944, 1043
bind time
after assembly time 1043
before execution time 1043
definition 1043
Binder 849
definition 1043
bit 19
data 358
definition 39, 1043
invert 354
naming problems 359
reset 354
set 354
test 356
bit bucket 244
bit-length constant 259
BL extended mnemonic 211
blank
definition 14, 1043, 1054
in constants 147, 150
not in self-defining terms 147, 150
representation in examples 13
text representation 14, 1043
Blefuscu 453
block comments 76
blocked records 967
BLR extended mnemonic 211
BM extended mnemonic 211
BMR extended mnemonic 211
BNE extended mnemonic 211
BNER extended mnemonic 211
BNH extended mnemonic 210
BNHR extended mnemonic 210
BNL extended mnemonic 210
BNLR extended mnemonic 210
BNM extended mnemonic 210
BNMR extended mnemonic 210
BNO extended mnemonic 210
BNOR extended mnemonic 210
BNP extended mnemonic 210
BNPR extended mnemonic 210
BNZ extended mnemonic 211
BNZR extended mnemonic 211
BO extended mnemonic 211
Boolean operations 288
BOR extended mnemonic 211
boundary alignment
See also alignment
by CNOP instruction 208
by DC/DS instruction 139
by ORG instruction 168
definition 145, 1043
BP extended mnemonic 211
BPR extended mnemonic 211
BR extended mnemonic 210
branch address 204, 305
and execute instruction 389
definition 213, 1043
branch condition 204
definition 213, 1043
branch mask 205
definition 213, 1043
branch relative 329
and execute instruction 393
and save 758
extended mnemonics 330
in place of branch on condition 330
on condition 329
on count 334
on index 340
BRAS machine instruction 758
BRASL machine instruction 758
BRC extended mnemonic 330
BRCT machine instruction 334
BRCTG machine instruction 334
BRE extended mnemonic 330
Breaking Event Address Register 212
BREL extended mnemonic 330
BRH extended mnemonic 330
BRHL extended mnemonic 330
BRL extended mnemonic 330
BRLL extended mnemonic 330
BRM extended mnemonic 330
BRML extended mnemonic 330
BRNE extended mnemonic 330
BRNEL extended mnemonic 330
BRNH extended mnemonic 330
BRNHL extended mnemonic 330
BRNL extended mnemonic 330
BRNLL extended mnemonic 330
BRNM extended mnemonic 330
BRNML extended mnemonic 330
BRNO extended mnemonic 330
BRNOL extended mnemonic 330
BRNP extended mnemonic 330
BRNPL extended mnemonic 330
BRNZ extended mnemonic 330
BRNZL extended mnemonic 330
BRO extended mnemonic 330
BROL extended mnemonic 330
BRP extended mnemonic 330
BRPL extended mnemonic 330
BRU extended mnemonic 330
BRUL extended mnemonic 330
BRXH machine instruction 341
BRXHG machine instruction 341
BRXLE machine instruction 341
BRXLG machine instruction 341
BRZ extended mnemonic 330
BRZL extended mnemonic 330
BSM machine instruction 858
BXH machine instruction 341
BXHG machine instruction 341
BXLE machine instruction 341
BXLEG machine instruction 341
byte 43
bit numbering 43
definition 48, 1043
memory address 43
byte reversal 453
BZ extended mnemonic 211
BZR extended mnemonic 211

C

C machine instruction 222
C-string 415
definition 427, 1043
destructive overlap
interruptible
null byte
calling point identifier 778
definition 788, 1043
CC
See Condition Code
CD machine instruction 615
CDB machine instruction 662
CDBR machine instruction 662
CDFBR machine instruction 666
CDFR machine instruction 620
CDGBR machine instruction 666
CDGR machine instruction 620
CDGTR machine instruction 707
CDGTRA machine instruction 707
CDR machine instruction 615
CDSTR machine instruction 709
CDTR machine instruction 705
CDUTR machine instruction 711
CDXT machine instruction 712
CDZT machine instruction 712
CE machine instruction 615
CEB machine instruction 662
CEBR machine instruction 662
CEDTR machine instruction 706
CEFBR machine instruction 666
CEFR machine instruction 620
CEGBR machine instruction 666
CEGR machine instruction 620
central processing unit (CPU) 42
access register 48
Condition Code 47
control register 48
definition 49, 1044
floating-point register 46
general register 45
general register pair 46
Instruction Address 47
Program Mask 47
Program Status Word 47
CER machine instruction 615
CEXTR machine instruction 706
CFDBR machine instruction 666
CFDR machine instruction 620
CFEBR machine instruction 666
CFER machine instruction 620
CFI machine instruction 322
CFXBR machine instruction 666
CFXR machine instruction 620
CG machine instruction 222
CGDBR machine instruction 666

CGDR machine instruction 620
CGDTR machine instruction 707
CGDTRA machine instruction 707
CGEBR machine instruction 666
CGER machine instruction 620
CGF machine instruction 230
CGFI machine instruction 322
CGFR machine instruction 230
CGHI machine instruction 322
CGR machine instruction 222
CGXBR machine instruction 666
CGXR machine instruction 620
CGXTR machine instruction 707
CGXTRA machine instruction 707
CH machine instruction 222
character
  ASCII 432
  BCD 429
  C-string 415
  decimal floating-point 687
  double-byte EBCDIC 434
  EBCDIC 430
  glyph 439
  not allowed in symbols 89
  representation
    ASCII 157, 1013
    EBCDIC 87, 1012
    Unicode 438
  shift-in 434
  shift-out 434
  Unicode 438
character self-defining term
  See self-defining term
characteristic
  binary floating-point 639
  decimal floating-point 687
  definition 585, 1043
  floating-point 584
  hexadecimal floating-point 586
CHI machine instruction 322
CHY machine instruction 222
CL machine instruction 232
class
  definition 867, 1044
class name 1046
CLC machine instruction 365, 378
CLCL machine instruction 404, 407
CLCLE machine instruction 404, 413
CLCLU machine instruction 441, 443
CLFI machine instruction 322
CLG machine instruction 232
CLGF machine instruction 232
CLGFI machine instruction 322
CLGFR machine instruction 232
CLGR machine instruction 232
CLI machine instruction 354
CLIY machine instruction 354
CLM machine instruction 232
CLMH machine instruction 232
CLMY machine instruction 232
CLOSE macro instruction 970
CLR machine instruction 232
CLST machine instruction 415, 419
CNOP assembler instruction 208
  definition 1044
CNOP instruction
  definition 213
code
  definition 83, 1044
code page
  definition 1044
CODEPAGE assembler option 439
coefficient 686
cohort 683
  definition 737, 1044
column order 910
  definition 944, 1044
column-major order
  definition 944, 1044
COM assembler instruction 803, 1044
combination field 686, 687
comment statement 74
common
  definition 867, 1044
common section 803, 1046
  COM assembler instruction 1044
  definition 1044
comparison
  binary floating-point 662, 663
  decimal floating-point 705
  decimal floating-point biased exponent 706
  fixed-point arithmetic 222
  fixed-point logical 232
  hexadecimal floating-point 615
  packed decimal 503
  register-immediate 322
  storage-immediate 354
  storage-storage 378
COMPAT(TRANSDT) assembler option 88, 433
complement addition
  definition 1044
  hexadecimal floating-point 613, 614
    process 613
  packed decimal 493
complement decimal addition
  definition 496, 1044
complementation
  fixed-point overflow 28
complex relocatability 99
  definition 99, 104, 1044
  term
    symbol attribute reference 104
complex relocatability attribute 132
COND= in macro operand 960
Condition Code 47, 234
  definition 49
  retrieve/set 234
conditional assembly
  conditional assembly language 1044
  definition 1044
conditional no-operation 207
  definition 213, 1044
constant
constant *(continued)*
  address constant 147
  alignment 137
  all types 1014
  ASCII 433
    translation 433
  binary 150
  binary floating-point
    decimal exponent 644
    exponent modifier 644
    length modifier 645
    rounding-mode suffix 643
    type DB 642
    type EB 642
    type LB 642
  binary floating-point symbolic operand
    (DMin) 643
    (Inf) 643
    (Max) 643
    (Min) 643
    (NaN) 643
    (QNaN) 643
    (SNaN) 643
  bit-length 259
  boundary alignment 139
  character 150
    type C 157
    type CA 157
    type CE 157
    type CU 157
  DBCS 436
  decimal exponent 143, 590, 644, 692
  decimal floating-point 690
    rounding-mode suffix 691
    type DD 690
    type ED 690
    type LD 690
  decimal floating-point symbolic operand
    (DMin) 690
    (Inf) 690
    (Max) 690
    (Min) 690
    (NaN) 690
    (QNaN) 690
    (SNaN) 690
  duplication factor 141
  embedded blanks 147, 150
  exponent modifier 144, 593, 644, 692
  fixed-point binary 146, 147
    unsigned 147
  floating-point
    summary 742
  hexadecimal 150
  hexadecimal floating-point 590, 595
    decimal exponent 591
    exponent modifier 593
    length modifier 592
    padding 592
    rounding mode suffix 595
    scale modifier 592
    truncation 592

constant *(continued)*
  hexadecimal floating-point *(continued)*
    type D 590
    type DH 590
    type E 590
    type EH 590
    type L 590
    type LH 590
    type LQ 594
  hexadecimal floating-point symbolic operand
    (DMin) 595
    (Max) 595
    (Min) 595
  length 137
  length attribute 143
  length modifier 140
  literal 154
  location-counter dependent 172
  multiple values 142
  nominal value 137
  offset 839
    Q-type 839
  packed decimal 467
  padding 152
  program modifier 138
  Q-type 839
  rounding-mode suffix summary 742
  S-type 149
  scaled fixed-point binary 555
  truncation 152
  type 137
  type A 147
  type AD 157
  type B 150
  type C 150
    ASCII characters 157
    EBCDIC characters 150
    Unicode characters 157
  type CA 157
  type CE 157
  type CU 157
  type D 567
  type DB 567
  type DD 567
  type DH 567, 594
  type E 567
  type EB 567
  type ED 567
  type EH 567, 594
  type extension 157
  type F 146
  type FD 146, 157
  type H 146
  type HD 146
  type L 567
  type LB 567
  type LD 567
  type LH 567, 594
  type LQ 594
  type S 149
  type V 820
  type X 150

constant *(continued)*
  type Y 149
  Unicode 439
  unsigned binary 147
  vs. self-defining term 150
  zero duplication factor 160
  zoned decimal 464
constant modifiers 138
constant type 138
  definition 145, 1044
continue column 75
control register 48
control section 803
  See also section
  blank 826
  COM 803
  common control section 1044
  CSECT 803, 1044
  definition 1044
  DSECT assembler instruction 804
  dummy control section 872
  DXD 804
  executable 803
  literals 808
  ordinary control section 1044
  Private Code 826
  reference 803
  resuming 807
  RSECT 803, 1044
  START 803
conversion
  among floating-point types 743
  BCD to declet 685
  between arbitrary bases 19
  between binary and hexadecimal 18
  between decimal and binary 20
  between packed and zoned decimal 469
  binary floating-point to integer 666
  decimal floating-point to integer 707
  decimal floating-point to signed packed decimal 709
  decimal floating-point to unsigned packed decimal 711
  declet to BCD 685
  fractions between bases 557
  hexadecimal floating-point to integer 621
  hexadecimal floating-point truncation 620
  In-Out 743
  integer to binary floating-point 666
  integer to decimal floating-point 707
  integer to hexadecimal floating-point 620
  Out-In 744
  signed packed decimal to decimal floating-point 709
  unsigned packed decimal to decimal floating-point 711

CONVERTI macro instruction 1016
  definition 1024
  description 1016
CONVERTO macro instruction 1017
  definition 1024
  description 1017
CP machine instruction 497, 503
CPSDR machine instruction 570, 700
CPU
  See central processing unit
CR machine instruction 222
CSDTR machine instruction 709
CSECT
  definition 1044
CSECT assembler instruction 803
CSECT instruction 803
CSXTR machine instruction 709
CU12 machine instruction 448
CU14 machine instruction 448
CU21 machine instruction 448
CU24 machine instruction 448
CU41 machine instruction 448
CU42 machine instruction 448
CUDTR machine instruction 711
CUSE machine instruction 415, 423
CUTFU machine instruction 448
CUUTF machine instruction 448
CUXTR machine instruction 711
CVB machine instruction 532, 534
CVBG machine instruction 532, 534
CVBY machine instruction 532, 534
CVD machine instruction 532
CVDG machine instruction 532, 533
CVDY machine instruction 532
CXBR machine instruction 662
CXD-type address constant
  definition 867, 1044
CXFBR machine instruction 666
CXFR machine instruction 620
CXGR machine instruction 620
CXGTR machine instruction 707
CXGTRA machine instruction 707
CXR machine instruction 615
CXSTR machine instruction 709
CXTR machine instruction 705
CXUTR machine instruction 711
CY machine instruction 222
CZDT machine instruction 712
CZXT machine instruction 712

D

D machine instruction 274, 275
DAT 67
  See also address translation
  Dynamic Address Translation 67
data
  access method 964
  access technique 966
  BDW 968
  BLKSIZE 967
  block descriptor word 968
  blocked records 967

data *(continued)*
BPAM 967
BSAM 966
Data Control Block 966
Data Definition Name 962
Data Set Name 962
DCB 966, 967, 968
DCB macro 966
DCBD macro 970
DCBE macro 971
DDName 962
direct access 971
DSName 962
DSORG 967
EODAD 966, 968
EXLST 966, 968
FIND 967
fixed length records 967
FREEPOOL macro 970
IHADCB dummy section 970
indexed access 971
JFCB 962
Job File Control Block 962
LRECL 967
MACRF 967
Partitioned Organization 967
QSAM 966
RDW 968
RECFM 967
record descriptor word 968
record formats 968
sequential access 966, 971
undefined length records 967
variable length records 967
VSAM 971
Data Control Block 963
data exception 57
Data Exception Code 650
binary floating-point
divide by zero 650
inexact 650
invalid operation 650
underflow 650
decimal floating-point
divide by zero 699
inexact 699
instruction availability 699
invalid operation 699
quantum exception 699
underflow 699
data groups
decimal floating-point 725
data structure
array 908
B-tree 940
doubly-linked list 934
DSECT 875
free storage list 929
hash table 941
list 927
mapping identical structures 895
mapping with DSECT 875
queue 934
stack 923, 924
tree 937
data structure *(continued)*
tree search 940
DBCS 434
constant 436
continuation rules 437
self-defining term 436
shift-in character 434
shift-out character 434
ward 435
DBCS assembler option 436
DC assembler instruction 137
DC statement 138
alignment 146
operands 138
delimiters 138
duplication factor 138
modifiers 138
nominal value 138
type 138
DCB 963
DCB macro instruction 966
DCB
DSORG 967
EODAD 966
EXLST 966
Partitioned Organization
FIND macro 967
library member 967
READ macro 967
WRITE macro 967
DCBD macro instruction 970
IHADCB dummy section 970
DCBE macro instruction 971
DD machine instruction 603
DDB machine instruction 659
DDBR machine instruction 659
DDR machine instruction 603
DDTR machine instruction 701
DDTRA machine instruction 701
DE machine instruction 603
DEB machine instruction 659
DEBR machine instruction 659
decimal data exception 485
definition 496, 1044
packed decimal arithmetic 496
decimal divide exception 57, 491
definition 496, 1044
decimal exponent 143
definition 145, 1044
in binary floating-point constant 644
in decimal floating-point constant 692
in fixed-point binary constant 143
in hexadecimal floating-point constant 591
decimal floating-point
addition/subtraction 703
binary-significand format 729
characteristic 687
coefficient 686
cohort 683
combination field 687
compare and signal 705
compare biased exponent 706
decimal floating-point *(continued)*
comparison 705
conceptual representation 682
constant 690
decimal exponent 692
exponent modifier 692
rounding-mode suffix 691
special values 690
type DD 690
type ED 690
type LD 690
convert BCD to declet 685
convert decimal floating-point to integer 707
convert decimal floating-point to signed packed decimal 709
convert decimal floating-point to unsigned packed decimal 711
convert declet to BCD 685
convert integer to decimal floating-point 707
convert signed packed decimal to decimal floating-point 709
convert to binary floating-point and hexadecimal floating-point 743
convert unsigned packed decimal to decimal floating-point 711
data classes
infinity 693
normal 693
QNaN 693
SNaN 693
subnormal 693
zero 693
data encoding 684
data formats 686
data groups 725
infinity or NaN 726
normal or subnormal, extreme exponent 726
normal, non-extreme exponent, nonzero leftmost digit 726
normal, non-extreme exponent, zero leftmost digit 726
zero, extreme exponent 726
zero, non-extreme exponent 726
declet 684
definition 1044
division 703
extract biased exponent 719
extract significance 720
floating-point integer 715
infinity 687
insert biased exponent 719
lengthening instructions 716
multiplication 702
NaN (Not a Number) 687
overview
exceptions 698
preferred encoding 685
preferred quantum 696
quantize 722
quantum 683
quantum exception 698

decimal floating-point *(continued)*
redundant representations 683
representation 686
options 681
properties 688
reround 724
rounding 695
rounding instructions 717
set rounding mode 718
shift significand 720
signed zero 749
significance of zero 749
test data class 693
test data group 726
TSF 686
ulp (unit in the last place) 683
zero 688
decimal overflow
See overflow
decimal overflow exception 57, 496, 1044
definition 496, 1044
packed decimal arithmetic 496
decimal self-defining term
See self-defining term
decimal specification exception 489
definition 1045
declet 684
definition 738, 1045
defined symbol
definition 95, 1045
definition
$$GENIO macro instruction 1028
A-type address constant 866
ABEND 981
abnormal termination 981
absolute symbol 104, 1041
adcon 158, 1041
addend 238, 1041
address constant 158, 866, 1041
address table 944, 1041
address translation 68, 1041
addressability 68, 133, 1041
addressability error 133, 1041
addressing halfword 69, 1042
addressing mode 315, 1042
algorithm 14, 1042
AMODE 315, 866, 1042
anchor 1042
AND operation 298, 1042
architecture 14, 1042
argument 788, 1042
arithmetic division 284, 1042
arithmetic multiplication 284, 1042
arithmetic representation 39, 1042
arithmetic shift 261, 1042
array 944, 1042
ASCII 1042
ASCII numeric character 483, 1042
assembler 14, 82, 1042
Assembler Language 14, 1042
definition *(continued)*
assembly time 82, 1042
augend 238, 1042
B-tree 944, 1042
base 69, 1043
base address 69, 1043
base register 69, 1043
base register specification digit 69, 1043
base_location 133
base_register 133
BCD 1043
BEAR 1043
bias 585, 1043
biased rounding 530, 1043
Big-Endian 1043
Binary Coded Decimal 1043
binary floating-point 1043
binary search 944, 1043
binary tree 944, 1043
Binder 1043
bit 39, 1043
blank 14, 1043
boundary alignment 145, 1043
branch address 213, 1043
branch condition 213, 1043
branch mask 213, 1043
Breaking Event Address Register 1043
byte 48, 1043
C-string 427, 1043, 1044
calling point identifier 788, 1043
Central Processing Unit 49, 1044
characteristic 585, 1043
class 867, 1044
CNOP assenbler instruction 1044
CNOP instruction 213
code 83, 1044
code page 1044
cohort 737, 1044
column order 944, 1044
column-major order 944, 1044
common 867, 1044
common section 1044
comparand 350, 1044
complement addition 1044
complement decimal addition 496, 1044
complex relocatability 104, 1044
Condition Code 49
conditional no-operation 213, 1044
constant type 145, 1044
control section 1044
CONVERTI macro instruction 1024
CONVERTO macro instruction 1024
CSECT 1044
CXD-type address constant 867, 1044
data dxception 1044
data exception 1044
Data Exception Code 677, 1044
DBCS 1045
definition *(continued)*
decimal data exception 1044
decimal divide exception 496, 1044
decimal exponent 145, 1044
decimal floating-point 1044
decimal overflow exception 496, 1044
decimal specification exception 496, 1045
declet 738, 1045
decode 1045
defined symbol 95, 1045
denormalization 585, 1045
denormalized number 677, 1045
densely packed decimal 738, 1045
Dependent USING 906, 1045
destructive overlap 427, 1045
DH displacement component 315, 1045
digit selector 548, 1045
digit selector and significance starter 549, 1054
diminished radix-complement representation 39, 1045
displacement 1045
20-bit 302
unsigned 12-bit 62
dividend 284, 1045
divisor 284, 1045
DL displacement component 315, 1045
Double-Byte Character Set 1045
double-ended queue 944, 1045
doubly-linked list 944, 1045
DPD 738, 1045
DROP assembler instruction 133, 1045
DSECT assembler instruction 906, 1045
dummy section 1045
DUMPOUT macro instruction 1025
duplication factor 145, 1045
DXC (Data Exception Code) 677, 1045
DXD 1045
EBCDIC 95, 1045
Effective Address 69, 1046
Effective Address Register 69, 1045
element 867, 1046
Encoded Length 115, 1046
entry point 788, 1046
entry point identifier 788, 1046
EQU extended syntax 174, 1046
Ex 1046
exception 1046
exception condition 59, 678, 1046, 1047
executable control section 867, 1046
execute 1046
execution time 83, 1046

definition *(continued)*
explicit address 115, 1046
explicit length 115, 1046
exponent 572, 1046
exponent modifier 145, 1046
exponent overflow 585, 1046
exponent underflow 585, 1046
expression 104, 1046
expression evaluation 104, 1046
extended mnemonic 213, 1046
external dummy 867, 1052
External Dummy Section 1046
External Symbol Dictionary 867, 1046
extreme exponent 738, 1046
field separator 548, 1046
fill character 548, 1047
floating-point 572, 1047
Floating-Point Control Register 678, 1047
Floating-Point Register 49, 1047
floating-point system FP(r,p) 572, 1047
floating-point system FPF(r,p) 572, 1047
floating-point system FPI(r,p) 572, 1047
free storage list 944, 1047
general register 49, 1047
GGn 202, 1047
glyph 1047
GOFF 867, 1051
gradual underflow 678, 1047
graphic data type 1047
GRn 202, 1047
guard digit 585, 1047
hash function 944, 1047
hash table 944, 1047
hexadecimal 39, 1047
High Level Assembler 1048
HLASM 14, 1047
ILC 49, 59
immediate operand 328, 1048
implied address 115, 1048
implied length 115, 1048
increment 350, 1048
index 69, 350, 1048
index register specification digit 69, 1048
indexing 69, 1048
infix notation 944, 1048
inorder tree traversal 944, 1048
insert 202, 1048
Instruction Address 1048
instruction cycle 1048
instruction decode 59
instruction execute 59
instruction fetch 59, 1046
Instruction Length Code 49, 59, 1048
instruction register 59, 397, 1048
interruptible 1048
interruption 59, 1048
invariant EBCDIC character 1048

definition *(continued)*
IR 397, 1048
Job Control Language 83, 1049
jump 1049
keyword argument 981
Labeled Dependent USING 906, 1049
Labeled USING 906, 1049
LC 1050
length attribute reference 104, 1049
Length Expression 115, 397, 1049
length modifier 145, 1049
Length Specification Byte 397
Length Specification Digit 1049
library 1049
linear subscript 944, 1049
linkage convention 788, 1049
linked list 944, 1049
Linker 83, 1049
linking loader 867, 1049
list 945, 1049
literal 158, 1049
literal pool 158, 1049
Little-Endian 1049
load module 83, 867, 1049
load operation 202, 1050
location counter (LC) 95, 1050
logical AND 1042
logical arithmetic 238, 1050
logical division 284, 1050
logical multiplication 284, 1050
logical operation 1050
logical OR 1051
logical representation 39, 1050
logical shift 261, 1050
logical XOR 1056
machine language 60, 1050
machine length 115, 1050
macro instruction 83, 1050
mantissa 572, 1050
mask 678, 1050
mask digit 1050
MaxReal 585, 1050
MBCS 1050
message character 549, 1050
millicode 49, 1050
MinReal 585, 1050
minuend 238, 1050
mnemonic 83, 115, 1050
modal instruction 315, 1050
modifier 145, 1050
Multiple-Byte Character Set 1050
multiplicand 284, 1050
multiplier 284, 1050
multiply and add/subtract 636, 1050
N' (Number attribute reference) 1051
N (Length Expression) 1051
NaN (Not a Number) 678, 1051
no-operation instruction 214, 1051
nominal value 145, 1051

definition *(continued)*
non-overflowed zero 496, 1051
normalization 585, 1051
null byte 1051
numeric digit 483, 1051
OBJ 867, 1051
object code 83, 1051
object module 83, 867, 1051
offset 530, 1051
ones' complement representation 39, 1051
opcode 115, 1051
operand 83, 1051
operand order dependence 530, 1051
operation code 60, 115, 1051
operator 15, 104, 1051
option 1051
OR operation 298, 1051
order dependence 496, 1051
Ordinary USING 906, 1051
ORG extended syntax 175, 1051
origin 83, 1051
overflow 39, 1051
overflowed zero 496, 1051
padding 158, 1051
parameter 788, 1051
parameterization 175, 1052
pattern character 549, 1052
payload 738, 1052
pipeline 214, 1052
PM 1052
positional argument 981
post-normalization 585, 1052
postfix notation 945, 1052
postorder tree traversal 945, 1052
pre-normalization 585, 1052
precision 530, 1052
preferred exponent 738, 1052
preferred quantum 738, 1052
preferred sign codes 483, 1052
preorder tree traversal 945, 1052
PRINTLIN macro instruction 1026
PRINTOUT macro instruction 1026
problem state 49, 1052
program interruption 981, 1052
program length 115, 1052
program linking 867, 1052
Program Loader 83, 1052
Program Mask 60, 1052
program object 867, 1052
Program Status Word 49, 1052
pseudoregister 867, 1052
Q-type address constant 867, 1052
QNaN 678, 1052
qualified symbol 906, 1052
qualifier 906, 1052
quantum 738, 1052
queue 945, 1052
quotient 284, 1052
R(r1+1) 1053
R(r3|1) 1053

definition *(continued)*
radix 572, 1053
radix-complement representation 39, 1053
READCARD macro instruction 1028
real address 69, 1053
recovery routine 981
recursion 992
reenterability 992
reenterable 1053
reentrant 993
reference control section 867, 1053
relative address 315, 1053
relocatable 95, 1053
relocate 867, 1053
relocating loader 867, 1053
relocation 83, 95, 867, 1053
relocation dictionary 868, 1053
remainder 284, 1053
return address 788, 1053
return code 788, 1053
RMODE 868, 1053
rotating shift 261, 1053
rounding digit 585, 1053
rounding mode 678, 1053
rounding modifier 678, 1054
rounding-mode suffix 636, 1054
row order 945, 1054
row-major order 945, 1054
RSECT 1054
SBCS 1054
scaled arithmetic 530, 1054
section 868, 1054
segment 868, 1054
self-defining term 95, 1054
Shift-In 1054
Shift-Out 1054
sign extension 203, 1054
sign-magnitude representation 39, 1054
significance exception 636, 1054
significance indicator 549, 1054
significance starter 549, 1054
significand 572, 1054
simple relocatability 104, 1054
Single-Byte Character Set 1054
SNaN 678, 1054
space 15, 1054
special value 678, 1054
stack 945, 1055
statement 83, 1055
statement field 84, 1055
status flag 678, 1055
store operation 203, 1055
subtrahend 238, 1055
supervisor state 49, 1055
symbol 95, 1055
symbol attribute 95, 1055
symbol attribute reference 104, 1055
Symbol Table 133, 1055
Syntactic Character Set 1055
system interruption 982

definition *(continued)*
system service 982
table 945, 1055
target instruction 398, 1055
term 104, 1055
text 868, 1055
true addition 1055
true decimal addition 496, 1055
truncation 158, 1055
two's complement 1055
two's complement representation 39, 1055
type extension 158, 1055
UCS 1055
ulp (unit in the last place) 585, 1055
unbiased rounding 530, 1055
Unicode 1055
Unicode numeric character 483, 1055
Unicode Transformation Format 1056
Universal Character Set 1055
unnormalized add/subtract 636, 1056
unnormalized number 1056
unsigned 12-bit displacement 69
USING assembler instruction 133, 1056
USING Table 134, 1056
UTF 1056
V-type address constant 868, 1056
virtual address 69, 1056
virtual origin 945, 1056
XOR operation 298, 1056
zero duplication factor 175, 1056
zero extension 203, 1056
zone digit 483, 1056
zoned digit 549
denormalization 647
definition 585, 1045
denormalized numbers 639
binary floating-point 647
densely packed decimal
definition 738, 1045
dependent USING 885, 904
anchor 1045
complex structures 885
definition 906, 1045
DROP assembler instruction 890
DER machine instruction 603
DIDBR machine instruction 669
DIEBR machine instruction 669
digit selector 537
definition 548, 1045
digit selector and significance starter 537
definition 549, 1054
diminished radix-complement representation 24
definition 39, 1045
discontinuity
Location Counter 808
displacement 62

displacement *(continued)*
20-bit 302
addressing halfword 1045
definition 69, 1045
unsigned 12-bit 62
dividend
binary division
by shifting 253
definition 284, 1045
fixed-point binary division 274
packed decimal 490
division
arithmetic 275
binary floating-point 659
decimal floating-point 703
double-length 275
fixed-point 274
fixed-point binary process 280
Fixed-Point Divide Exception 274
floating-point 578
hexadecimal floating-point 603
logical 279
packed decimal 490, 509
register pair 274
single-length 278
divisor
definition 284, 1045
fixed-point binary division 274
packed decimal 490
DL machine instruction 274, 279
DLG machine instruction 274, 279
DLGR machine instruction 274, 279
DLR machine instruction 274, 279
Double-Byte Character Set
definition 1045
double-byte EBCDIC 434
shift-in 434
shift-out 434
ward 435
double-ended queue
definition 944, 1045
doubleword 44, 177
doubly-linked list 934
definition 944, 1045
DP machine instruction 497, 509
DPD
definition 738, 1045
DR machine instruction 274, 275
DROP assembler instruction 128, 883, 890, 905
definition 133, 1045
dependent USING 890, 905
labeled dependent USING 905
labeled USING 883, 905
ordinary USING 128, 905
summary 905
DS assembler instruction 159
DSECT assembler instruction 804, 872, 874
address resolution 874
as External Dummy Section 838
DCBD 970
definition 906, 1045
EPIE 975

DSECT assembler instruction *(continued)*
Location Counter 873
multiple data structures 875
named in Q-type address constant 838
relocation attribute 873
DSG machine instruction 274, 275
DSGF machine instruction 275
DSGFR machine instruction 275
DSGR machine instruction 274, 275
Dummy Control Section
See DSECT assembler instruction
dummy external section 1046
dummy section
DSECT instruction 1045
DXD instruction 1045
DUMPOUT macro instruction 1018
definition 1025
description 1018
duplicate definition 123
duplication factor 138
default 138
definition 145, 1045
DXBR machine instruction 659
DXC 649
See also Data Exception Code
DXD assembler instruction 804
DXR machine instruction 603
DXTR machine instruction 701
DXTRA machine instruction 701

E

EBCDIC 430, 434
C-type constants 150
code pages 430
definition 95, 1045
double-byte 434
in character self-defining terms 86
table 1012
ED machine instruction 532, 538
editing 536
flow diagram 547
overview 536
pattern 536
process 539
zoned digit 546
EDMK machine instruction 532, 543
EEDTR machine instruction 719
EEXTR machine instruction 719
Effective Address 62
24-bit addressing mode 308
31-bit addressing mode 308
64-bit addressing mode 308
definition 69, 1046
relative-immediate 305
signed 20-bit displacement 302
unsigned 12-bit displacement 302
Effective Address Register 62
definition 69, 1045
EFPC machine instruction 651
element
definition 867, 1046
Encoded Length 366
definition 115, 1046
Length Specification Byte 366
machine length 366
end column 75
END record in object module 832
ENTRY assembler instruction 821, 1046
entry point 765
definition 788, 1046
entry point identifier 777
definition 788, 1046
EPICA
See Program Interruption Control Area
EPIE
See Program Interruption Element
EQU assembler instruction 93, 162
extended syntax 166, 1046
definition 174
ESD record in object module 831
CM symbol type 826
CQ symbol type 826
ER symbol type 826
LD symbol type 826
PC symbol type 826
PQ symbol type 826
SD symbol type 826
SQ symbol type 826
WX symbol type 826
XD symbol type 826
ESDID 818, 831
ESDTR machine instruction 720
ESPIE macro instruction 973
program interruption exit 973
ESTAE macro
recovery routine 977
retry routine 977
ESTAE macro instruction 977, 978
ESTAEX macro instruction 977
ESXTR machine instruction 720
EX machine instruction 389
exception 56
See also interruption
binary floating-point 649
division by zero 649
exponent overflow 649
exponent underflow 649
inexact result 649
invalid operation 649
signaling NaN 641
decimal floating-point
divide by zero 699
inexact 699
instruction availability 699
invalid operation 699
quantum 698, 699
underflow 699
definition 1046
hexadecimal floating-point
division by zero 603
exponent overflow 602
exponent underflow 602
lost significance 607
exception action
exception action *(continued)*
binary floating-point
divide by zero 652
exponent overflow 652
exponent underflow 652
inexact result 652
invalid operation 651
exception condition 58
definition 59, 1046
interruption if enabled 58
executable control section 803
CSECT 803
definition 867, 1046
RSECT 803
START 803
execute exception 57
execute instructions 389
execution time 74
address 92
definition 83, 1046
explicit address 109, 110, 367
definition 115, 1046
explicit length 113, 368
definition 115, 1046
in constants 140
exponent
binary floating-point 639
decimal floating-point 687
definition 572, 1046
hexadecimal floating-point 586
exponent modifier 143
binary floating-point constant 644
decimal floating-point constant 692
definition 145, 1046
hexadecimal floating-point constant 593
exponent of a floating-point number
definition 1046
exponent overflow 582
binary floating-point 649
definition 585, 1046
hexadecimal floating-point 602
exponent underflow 582
binary floating-point 649
definition 585, 1046
hexadecimal floating-point 602
expression 97
absolute 99
complexly relocatable 99
definition 104, 1046
evaluation 98, 103
definition 104
factor 103
operator
− 97
/ 97
* 97
+ 97
unary − 97
unary + 97
operator precedence 98
paired terms 99
parentheses 97
primary 103

expression *(continued)*
products 98
quotients 98
relocatable terms 98
simply relocatable 99
term 104
unpaired term 99
EXRL machine instruction 389
extended mnemonic 210
B 210
BE 211
BER 211
BH 211
BHR 211
BL 211
BLR 211
BM 211
BMR 211
BNE 211
BNER 211
BNH 210
BNHR 210
BNL 210
BNLR 210
BNM 210
BNMR 210
BNO 210
BNOR 210
BNP 210
BNPR 210
BNZ 211
BNZR 211
BO 211
BOR 211
BP 211
BPR 211
BR 210
BRC 330
BRE 330
BREL 330
BRH 330
BRHL 330
BRL 330
BRLL 330
BRM 330
BRML 330
BRNE 330
BRNEL 330
BRNH 330
BRNHL 330
BRNL 330
BRNLL 330
BRNM 330
BRNML 330
BRNO 330
BRNOL 330
BRNP 330
BRNPL 330
BRNZ 330
BRNZL 330
BRO 330
BROL 330
BRP 330
BRPL 330
BRU 330
extended mnemonic *(continued)*
BRUL 330
BRZ 330
BRZL 330
BZ 211
BZR 211
definition 213, 1046
J 330
JC 330
JCT 334
JCTG 334
JE 330
JH 330
JL 330
JLC 330
JLE 330
JLH 330
JLL 330
JLM 330
JLNE 330
JLNH 330
JLNL 330
JLNM 330
JLNO 330
JLNOP 330
JLNP 330
JLNZ 330
JLO 330
JLP 330
JLU 330
JLZ 330
JM 330
JNE 330
JNH 330
JNL 330
JNM 330
JNO 330
JNOP 330
JNP 330
JNZ 330
JO 330
JP 330
jump instructions 330
JXH 343
JXHG 343
JXLE 343
JXLEG 343
JZ 330
NOP 206, 211
NOPR 206, 211
extended save area conventions 773
external dummy section 804
definition 867, 1052
External Dummy Section.
definition 1046
external symbol 818, 1046
ALIAS assembler instruction 1046
class name 1046
COM 818
common section 1046
control section name 1046
CSECT 818
definition 1046
DSECT
named in Q-type address constant 838
external symbol *(continued)*
DSECT assembler instruction 818
dummy external section 1046
DXD 1046
DXD assembler instruction 818
ENTRY 818
ENTRY assembler instruction 1046
EXTRN 818
EXTRN statement 1046
pseudoregister 1046
renaming via ALIAS statement 1046
RSECT 818
START 818
WXTRN 818
WXTRN statement 1046
External Symbol Dictionary 93
definition 867, 1046
ESDID 818
Owning ID 818
relocation attribute 825
extreme exponent
binary floating-point 649
decimal floating-point 699
definition 738, 1046
hexadecimal floating-point 602
EXTRN assembler instruction 818, 820
difference from WXTRN 820
EXTRN statement 1046

F

F-format records 967
FIDBR machine instruction 668
FIDR machine instruction 625
FIDTR machine instruction 715
FIEBR machine instruction 668
field separator 537
definition 548, 1046
FIER machine instruction 625
fill character 537
definition 548, 1047
FIND macro 967
DCB
MACRF 967
FIXBR machine instruction 668
fixed length records 967, 968
fixed-point binary
arithmetic division 275
compare instructions 232
comparison 222
divide exception 57
division 274
double-length division 275
double-length multiplication 265
logical division 279
logical multiplication 270
mixed integer-fraction representation 554
multiplication 264
shifting 252
signed multiplication 265

fixed-point binary *(continued)*
single-length division 278
single-length multiplication 267
fixed-point overflow 28
See also overflow
binary complementation 28
two's complement 31
fixed-point overflow exception 57
FIXR machine instruction 625
FIXTR machine instruction 715
flag bit
Floating-Point Control Register 649
floating-point
as rational numbers 562
binary 1043
constant
type D 567
type DB 567
type DD 567
type DH 594, 567
type E 567
type EB 567
type ED 567
type EH 594, 567
type L 567
type LB 567
type LD 567
type LH 594, 567
type LQ 594
constants summary 742
convert among types 743
decimal 1044
definition 572, 1047
exceptions summary 741
GPR-FPR copying instructions 569
load zero instructions 569
MaxReal 582
MinReal 582
overview 560, 743
addition/subtraction 578
base 560
bias 584
characteristic 584
division 578
exponent 560
exponent overflow 582
exponent representation 561
exponent sign 561
exponent underflow 582
exponent width 560
FP(r,p) 743
FPF(r,p) 563
FPI(r,p) 563
FPN(r,p) 563
guard digit 575
multiplication 573
normalization 573
post-normalization 573
pre-normalization 574
radix 560
representation 560
rounding 563
rounding digit 575
significand 560
floating-point *(continued)*
properties summary 741
real vs. realistic arithmetic 745
representation examples 749
representation summary 739
representation-independent instructions 568, 699
rounding-mode suffix summary 742
sign-copying instructions 570
summary 739
System z
base 10 564
base 16 564
base 2 564
floating-point register pairs 565
floating-point registers 564
ulp (unit in the last place) 581
zero behavior 747
Floating-Point Control Register 649
binary floating-point 649
binary rounding mode 650
decimal floating-point 701, 718
decimal rounding mode 718
DXC 649
instructions 651
floating-point guard digit 575
floating-point overflow
See overflow
floating-point register 46, 564
floating-point rounding digit 575
floating-point summary
constant rounding-mode suffix 742
constants 742
exceptions 741
properties 741
representations 739
floating-point system FP(r,p) 562
definition 572, 1047
floating-point system FPF(r,p) 563
definition 572, 1047
floating-point system FPI(r,p) 563
definition 572, 1047
floating-point system FPN(r,p) 563
floating-point underflow
See underflow
FPCR
See also Floating-Point Control Register
definition 1047
FPR
See also floating-point register
definition 49, 1047
fraction conversion
between bases 557
free storage list 929
definition 944, 1047
FREEMAIN macro instruction 957, 959, 961
FREEPOOL macro instruction 970
fullword 44

G

general purpose register
definition 1047
general register
definition 49, 1047
pair 46
register halves 178
generalized object file format
definition 1047
object module 1047
GET macro instruction 964
GETMAIN macro instruction 957, 958, 961
GGn 190
definition 202, 1047
Glyph
definition 1047
GOFF
See generalized object file format
GOFF object module
definition 867, 1051
GOFF option
definition 1047
GPR
See general purpose register
See general register
gradual underflow 647
graphic data type
See also DBCS
definition 1047
GRn 179, 190
definition 202, 1047
guard digit 575
definition 585, 1047

H

halfword 44, 177
halfword operands 182
halve instructions
hexadecimal floating-point 604
hash function 941
definition 944, 1047
hash table 941, 942
definition 944, 1047
HDR machine instruction 604
HER machine instruction 604
hex 19
See also hexadecimal
abbreviation for hexadecimal 19
hexadecimal
definition 39, 1047
digits 18
hexadecimal floating-point 586
addition/subtraction 606
no pre-normalization 606
characteristic 586
comparison 615
constant 590
conversion instructions
to/from fixed binary 620
convert to binary floating-point and decimal floating-point 743
data representation 586

hexadecimal floating-point *(continued)*
division 603
exponent 586
exponent overflow 602
exponent underflow 602, 612
halve 604
history 629
integer values 625
lengthening instructions 618
lost significance exception 607
more precise rounding 595
multiplication 599
multiply and add/subtract 627
normalized addition 606
normalized subtraction 606
post-normalization 599, 603
pre-normalization 599, 603
pseudo-zero 748
remainder 625
rounding instructions 616
square root 626
subtype H 595
symbolic operand
(DMin) 595
(Max) 595
(Min) 595
truncation in conversion 620
unnormalized addition 609
unnormalized representation 588
unnormalized subtraction 609
hexadecimal floating-point divide 57
hexadecimal floating-point exponent overflow 57
hexadecimal floating-point exponent underflow 57
hexadecimal floating-point lost significance 57
hexadecimal self-defining term
See self-defining term
High Level Assembler
definition 1047, 1048
HLASM 4
definition 14, 1047

I

IA
See Instruction Address
IARV64 macro instruction 957
IBM High Level Assembler for z/OS & z/VM & z/VSE 4
IC machine instruction 184
ICM machine instruction 185
ICMH machine instruction 189
IEDTR machine instruction 719
IEXTR machine instruction 719
IHADCB dummy section 970
IHADCB macro instruction 970
IHAEPIE dummy section 975
IHAEPIE macro instruction 975
IIHF machine instruction 318
IIHH machine instruction 318
IIHL machine instruction 318
IILF machine instruction 318
IILH machine instruction 318
IILL machine instruction 318
ILC
See also Instruction Length Code
definition 1048
immediate operand
add 321
arithmetic load 318
compare 322
logical AND 323, 354
logical insert 318
logical load 318, 321
logical OR 323, 354
logical XOR 324, 354
move 353
multiply 322
subtract 321
test 356
implied address 109, 110
absolute 131
base register zero 131
definition 115, 1048
implied length 113, 368
definition 115, 1048
in constants 140
in SS-type instructions 367
increment
definition 1048
index 63
definition 69, 1048
index register
definition 1048
general purpose register 1048
index register specification digit 1048
index register specification digit 63
definition 69, 1048
indexed Effective Address 64
indexing 63
definition 69, 1048
operations 332
inexact result
binary floating-point 649
decimal floating-point 698
infinity
binary floating-point 641
decimal floating-point 687
infix notation 923
definition 944, 1048
inorder tree traversal 940
definition 944, 1048
input/output
access technique
direct 966
indexed 966
sequential 966
CLOSE macro 970
FREEPOOL macro 970
GET macro 964
OPEN macro 969
PUT macro 966
QSAM
locate mode 966
move mode 966
sequential access
basic 966
input/output *(continued)*
sequential access *(continued)*
BSAM 966
QSAM 966
queued 966
VSAM 971
insert 184
character 184
definition 202, 1048
logical-immediate 318
insert under mask
under mask 185
instruction
addressing halfword 53
basic types
RR 51
RS 51
RX 51
SI 51
SS 51
cycle 50
halfword alignment 50
Instruction Address 50
Instruction Length Code 53
instruction register 50
length 50
modal 308
operand address 53
operation code 52
operation code examples 54
partial completion 403
pipeline 207
Instruction Address 47, 50
definition 1048
updated 50
vs. location counter 92
instruction cycle
decode 51, 1045, 1048
definition 59
definition 1048
execute 51, 1046, 1048
definition 59
fetch 50, 1048
definition 59
with interruptions 56
instruction format
BRC 329
BRCL 329
RI-type 317
RIL-type 317
RRE-type 192, 415
RRF-type 444
RS-type 180, 186
RS-type shift 243
RSY-type 190, 410
RX-type 179
RXY-type 190
SI-type 352
SIY-type 352
SS-type, single-length 366
SS-type, two-length 469
Instruction Length Code 47, 53
definition 49, 59, 1048
instruction modification 361
instruction operation

instruction operation *(continued)*
CLCL 407
CLCLE 413
CLST 419
EX 389
EXRL 389
MVC 372
MVCL 405
MVCLE 411
MVST 417
SRST 416
TR 379
TRT 384
TRTR 387
instruction register 50
definition 59, 397, 1048
integer conversion
between bases 19
internal symbol
Assembler Language 1048
definition 1048
SYSADATA file 1048
internal symbol dictionary
See also symbol table
definition 1048
interruptible
definition 1048
interruptible instruction 403
interruption 55
asynchronous 56
binary floating-point division by zero 649
binary floating-point exponent overflow 649
binary floating-point exponent underflow 649
binary floating-point inexact result 649
binary floating-point invalid operation 649
binary floating-point rounding instructions 664
classes 56
code 56
data exception 485
decimal divide 57, 491
decimal floating-point division by zero 698
decimal floating-point exponent overflow 698
decimal floating-point exponent underflow 698
decimal floating-point inexact result 698
decimal floating-point invalid operation 698
decimal floating-point quantum exception 698
decimal floating-pointquantum exception 698
decimal overflow 57, 485
definition 59, 1048
disabled 56
enabled 56
ESPIE macro 973
interruption *(continued)*
exception 58
execute 389
execute exception 57
fixed-point divide 57
fixed-point overflow 57
hexadecimal floating-point divide 57, 603
hexadecimal floating-point exponent overflow 57, 602, 617
hexadecimal floating-point exponent underflow 57, 58, 602
hexadecimal floating-point lost significance 57, 58, 607
hexadecimal floating-point square root 627
invalid operation 57
involuntary 56
maskable 974
packed decimal division 491
packed decimal multiplication 489
PIE 973
privileged operation 57
program 56, 974
program interruption exit 973
Program Mask 58
specification exception 489, 491
SPIE macro 973
SVC 950
synchronous 56
voluntary 56
interruption code 56
1 (invalid operation) 57
10 (decimal overflow) 57, 485
11 (decimal divide) 57, 491
12 (hexadecimal floating-point exponent overflow) 602, 57, 617
13 (hexadecimal floating-point exponent underflow) 602, 57, 58
14 (hexadecimal floating-point lost significance) 57, 58, 607
15 (hexadecimal floating-point divide) 57, 603
2 (privileged operation) 57
29 (hexadecimal floating-point square root) 627
3 (execute exception) 57, 389
4 (access exception) 57
5 (addressing) 57
6 (specification) 57, 489, 491
7 (binary floating-point exception) 650
7 (data exception) 57, 485
7 (decimal floating-point exception) 699
8 (fixed-point overflow) 57
9 (fixed-point divide) 57, 274
definition 59, 1048
invalid operation exception 57
invariant EBCDIC character
definition 1048
IPM machine instruction 234
IR
See also Instruction Register
IR *(continued)*
definition 1048

J

J extended mnemonic 330
JC extended mnemonic 330
JCL 962
JCT extended mnemonic 334
JCTG extended mnemonic 334
JE extended mnemonic 330
JFCB 962
JH extended mnemonic 330
JL extended mnemonic 330
JLC extended mnemonic 330
JLE extended mnemonic 330
JLH extended mnemonic 330
JLL extended mnemonic 330
JLM extended mnemonic 330
JLNE extended mnemonic 330
JLNH extended mnemonic 330
JLNL extended mnemonic 330
JLNM extended mnemonic 330
JLNO extended mnemonic 330
JLNOP extended mnemonic 330
JLNP extended mnemonic 330
JLNZ extended mnemonic 330
JLO extended mnemonic 330
JLP extended mnemonic 330
JLU extended mnemonic 330
JLZ extended mnemonic 330
JM extended mnemonic 330
JNE extended mnemonic 330
JNH extended mnemonic 330
JNL extended mnemonic 330
JNM extended mnemonic 330
JNO extended mnemonic 330
JNOP extended mnemonic 330
JNP extended mnemonic 330
JNZ extended mnemonic 330
JO extended mnemonic 330
Job Control Language 81, 962
definition 83, 1049
Job File Control Block 962
JP extended mnemonic 330
jump
definition 1049
jump extended mnemonic 330
jump instructions 330
JXH extended mnemonic 343
JXHG extended mnemonic 343
JXLE extended mnemonic 343
JXLEG extended mnemonic 343
JZ extended mnemonic 330

K

KDB machine instruction 663
KDBR machine instruction 663
KDTR machine instruction 705
KEB machine instruction 663
KEBR machine instruction 663
keyword argument
definition 981

KXBR machine instruction 663
KXTR machine instruction 705

L

L machine instruction 179
LA machine instruction 309
label
 definition 1049
 labeled USING statement 1049
 name field symbol 1049
 qualifier 1049
labeled dependent USING 891, 904
 complex data structures 891
 definition 906, 1049
 identical data structures 895
labeled USING 882, 904
 address resolution 882
 definition 906, 1049
 DROP assembler instruction 883
large programs 790
 address constants 793
 addressability 790
  uniform addressability 790
LARL machine instruction 309
LAY machine instruction 309
LB machine instruction 196
LBR machine instruction 196
LC
 definition 1050
LCDBR machine instruction 655
LCDFR machine instruction 568
LCDR machine instruction 597
LCDTR machine instruction 715
LCEBR machine instruction 655
LCER machine instruction 597
LCGFR machine instruction 194
LCGR machine instruction 192
LCR machine instruction 187
LCXBR machine instruction 655
LCXR machine instruction 597
LD machine instruction 568
LDE machine instruction 619
LDEB machine instruction 665
LDEBR machine instruction 665
LDER machine instruction 619
LDETR machine instruction 716
LDGR machine instruction 570, 700
LDR machine instruction 568, 700
LDXBR machine instruction 664
LDXR machine instruction 616
LDXTR machine instruction 717
LDY machine instruction 568
LE machine instruction 568
LEDBR machine instruction 664
LEDR machine instruction 616
LEDTR machine instruction 717
length attribute 90
length attribute reference 368
 definition 104, 1049
Length Expression 366, 369
 definition 115, 1049
 explicit 366
 implied 366
length field 113, 469
length field *(continued)*
 explicit 113, 469
 implied 113, 469
 single-length instruction 113
 two-length instruction 114
length modifier 139, 259
 binary floating-point constant 645
 bit 259
 byte 140, 259
 decimal floating-point constant 692
 definition 145, 1049
 fixed-point binary 140
 hexadecimal floating-point constant 592
 packed decimal constant 467
 zoned decimal constant 464
Length Specification Byte 366, 370
 definition 397, 1049
 relation to Length Expression 370
Length Specification Digit
 definition 1049
lengthening instructions
 binary floating-point 664
 decimal floating-point 716
 hexadecimal floating-point 618
LER machine instruction 568, 700
LEXBR machine instruction 664
LEXR machine instruction 616
LEY machine instruction 568
LFAS machine instruction 651
LFPC machine instruction 651
LG machine instruction 189
LGB machine instruction 196
LGBR machine instruction 196
LGDR machine instruction 570, 700
LGF machine instruction 194
LGFI machine instruction 319
LGFR machine instruction 194
LGH machine instruction 189
LGHI machine instruction 319
LGHR machine instruction 192
LGR machine instruction 192
LH machine instruction 182
LHI machine instruction 319
LHR machine instruction 192
library
 definition 1049
 load module 846
 PDS 845
Lilliput 453
linear subscript 911
 definition 944, 1049
linkage 757
linkage convention 765
 argument 765
 argument address list 766
 argument passing
  32-bit addresses 766
 assisted linkage 783, 847
 calling point identifier 778
 definition 788, 1049
 entry point 765
 entry point identifier 777
 parameter 765
linkage convention *(continued)*
 return address 765
 return code 779
 return flag 778
 RETURN macro instruction 781
 save area 770
 SAVE macro instruction 777
 subroutine 765
 V-type address constant 820
 variable-length argument list 767
linked list 927
 See also list
 definition 944, 1049
Linker 73
 Binder 849
 boundary alignment 139, 809
 common sections 836
 Composite External Symbol Dictionary 835
 control sections 802
 definition 83, 1049
 library search 820
 load module 845
 options 984
linking
 See program linking
linking loader
 definition 867, 1049
list 927
 as an array 908
 definition 945, 1049
 deletion 929
 doubly-linked 934
 insertion 927
literal 154
 as a term 96
 definition 158, 1049
 in multiple control sections 808, 854
 in Private Code section 826
 rules 154
 special symbol 154
literal pool 156
 definition 158, 1049
Little-Endian 453
 definition 1049
LLC machine instruction 196
LLCR machine instruction 196
LLGC machine instruction 196
LLGCR machine instruction 196
LLGF machine instruction 196
LLGFR machine instruction 196
LLGH machine instruction 196
LLGHR machine instruction 196
LLGT machine instruction 196, 863
LLGTR machine instruction 196, 863
LLH machine instruction 196
LLHR machine instruction 196
LLIHF machine instruction 319
LLIHH machine instruction 319
LLIHL machine instruction 319
LLILF machine instruction 319
LLILH machine instruction 319
LLILL machine instruction 319

LM machine instruction 180
LMG machine instruction 189
LMH machine instruction 189
LNDBR machine instruction 655
LNDFR machine instruction 568
LNDR machine instruction 597
LNDTR machine instruction 715
LNEBR machine instruction 655
LNER machine instruction 597
LNGFR machine instruction 194
LNGR machine instruction 192
LNR machine instruction 187
LNXBR machine instruction 655
LNXR machine instruction 597
load immediate
  arithmetic 318
  logical 318
  with IILF 318
load module 73
  CESD record 846
  creation 73
  CTL/RLD record 846
  definition 83, 867, 1049
  EOM record 846
  IDR record 846
  loaded into memory 73
  Partitioned Data Set 846
  PDS 846
  RLD record 846
  SYM record 846
  TEXT record 846
load operation 179
  definition 202, 1050
location 92
  assembly time 92, 1050
  base location 1043
  definition 1050
  execution time address 1050
  location counter 1050
  vs. address 92
Location Counter 92
  definition 95, 1050
  discontinuity 808
  Location Counter values and LOCTR 810
  reference 121
  symbol definition 123
  threading 809
  vs. Instruction Address 92
location counter reference 97, 121
location vs. address 92
  assembly vs. execution time 92
LOCTR group
logical
  AND operation 289, 290
    definition 298
  Boolean 288
  differences vs. arithmetic 37
  operations 288
  OR operation 289, 291
    definition 298
  XOR operation 289, 292
    definition 298
logical AND
  definition 1042
logical arithmetic 224
  definition 238, 1050
logical division 279
  definition 284, 1050
logical multiplication 270
  definition 284, 1050
logical operation
  AND 1042
  definition 1050
  OR 1051
  XOR 1056
logical OR
  definition 1051
logical representation 37
  definition 39, 1050
logical shift 243
  definition 261, 1050
  double-length 248
  process 243
  single-length 245
logical XOR
  definition 1056
LPDBR machine instruction 655
LPDFR machine instruction 568
LPDR machine instruction 597
LPDTR machine instruction 715
LPEBR machine instruction 655
LPER machine instruction 597
LPGFR machine instruction 194
LPGR machine instruction 192
LPR machine instruction 187
LPXBR machine instruction 655
LPXR machine instruction 597
LR machine instruction 187
LRDR machine instruction 616
LRER machine instruction 616
LRV machine instruction 453
LRVG machine instruction 453
LRVGR machine instruction 453
LRVH machine instruction 453
LRVR machine instruction 453
LT machine instruction 194
LTDBR machine instruction 655
LTDR machine instruction 597
LTDTR machine instruction 715
LTEBR machine instruction 655
LTER machine instruction 597
LTG machine instruction 194
LTGFR machine instruction 194
LTGR machine instruction 192
LTORG
  in multiple control sections 808
LTORG assembler instruction 156
LTR machine instruction 187
LTXBR machine instruction 655
LTXR machine instruction 597
LXD machine instruction 619
LXDB machine instruction 665
LXDBR machine instruction 665
LXDR machine instruction 619
LXDTR machine instruction 715, 716
LXE machine instruction 619
LXEB machine instruction 665
LXEBR machine instruction 665
LXER machine instruction 619
LXR machine instruction 568, 700
LZDR machine instruction 569
LZER machine instruction 569
LZXR machine instruction 569

M

M machine instruction 264, 265
machine instruction
  A 216
  AD 606
  ADB 661
  ADBR 661
  ADR 606
  ADTR 701
  ADTRA 701
  AE 606
  AEB 661
  AEBR 661
  AER 606
  AFI 321
  AG 216
  AGF 230
  AGFI 321
  AGFR 230
  AGHI 321
  AGR 216
  AH 216
  AHI 321
  AL 224
  ALC 228
  ALCG 228
  ALCGR 228
  ALCR 228
  ALFI 321
  ALG 224
  ALGF 230
  ALGFI 321
  ALGFR 230
  ALGR 224
  ALR 224
  AP 497, 501
  AR 216
  AXBR 661
  AXR 606
  AXTR 701
  AXTRA 701
  BAS 758
  BASR 758
  BASSM 858
  BC 204, 205
  BCR 204, 205
  BCT 334
  BCTG 334
  BCTGR 334
  BCTR 334
  BRAS 758
  BRASL 758
  BRCT 334
  BRCTG 334
  BRXH 341
  BRXHG 341
  BRXLE 341
  BRXLG 341

machine instruction *(continued)*
BSM 858
BXH 341
BXHG 341
BXLE 341
BXLEG 341
C 222
CD 615
CDB 662
CDBR 662
CDFBR 666
CDFR 620
CDGBR 666
CDGR 620
CDGTR 707
CDGTRA 707
CDR 615
CDSTR 709
CDTR 705
CDUTR 711
CDXT 712
CDZT 712
CE 615
CEB 662
CEBR 662
CEDTR 706
CEFBR 666
CEFR 620
CEGBR 666
CEGR 620
CER 615
CEXTR 706
CFDBR 666
CFDR 620
CFEBR 666
CFER 620
CFI 322
CFXBR 666
CFXR 620
CG 222
CGDBR 666
CGDR 620
CGDTR 707
CGDTRA 707
CGEBR 666
CGER 620
CGF 230
CGFI 322
CGFR 230
CGHI 322
CGR 222
CGXBR 666
CGXR 620
CGXTR 707
CGXTRA 707
CH 222
CHI 322
CHY 222
CL 232
CLC 365, 378
CLCL 404, 407
CLCLE 404, 413
CLCLU 441, 443
CLFI 322
CLG 232

machine instruction *(continued)*
CLGF 232
CLGFI 322
CLGFR 232
CLGR 232
CLI 354
CLIY 354
CLM 232
CLMH 232
CLMY 232
CLR 232
CLST 415, 419
CP 497, 503
CPSDR 570, 700
CR 222
CSDTR 709
CSXTR 709
CU12 448
CU14 448
CU21 448
CU24 448
CU41 448
CU42 448
CUDTR 711
CUSE 415, 423
CUTFU 448
CUUTF 448
CUXTR 711
CVB 532, 534
CVBG 532, 534
CVBY 532, 534
CVD 532
CVDG 532, 533
CVDY 532
CXBR 662
CXFBR 666
CXFR 620
CXGR 620
CXGTR 707
CXGTRA 707
CXR 615
CXSTR 709
CXTR 705
CXUTR 711
CY 222
CZDT 712
CZXT 712
D 274, 275
DD 603
DDB 659
DDBR 659
DDR 603
DDTR 701
DDTRA 701
DE 603
DEB 659
DEBR 659
DER 603
DIDBR 669
DIEBR 669
DL 274, 279
DLG 274, 279
DLGR 274, 279
DLR 274, 279
DP 497, 509

machine instruction *(continued)*
DR 274, 275
DSG 274, 275
DSGF 275
DSGFR 275
DSGR 274, 275
DXBR 659
DXR 603
DXTR 701
DXTRA 701
ED 532, 538
EDMK 532, 543
EEDTR 719
EEXTR 719
EFPC 651
ESDTR 720
ESXTR 720
EX 389
EXRL 389
FIDBR 668
FIDR 625
FIDTR 715
FIEBR 668
FIER 625
FIXBR 668
FIXR 625
FIXTR 715
HDR 604
HER 604
IC 184
ICM 185
ICMH 189
IEDTR 719
IEXTR 719
IIHF 318
IIHH 318
IIHL 318
IILF 318
IILH 318
IILL 318
IPM 234
KDB 663
KDBR 663
KDTR 705
KEB 663
KEBR 663
KXBR 663
KXTR 705
L 179
LA 309
LARL 309
LAY 309
LB 196
LBR 196
LCDBR 655
LCDFR 568
LCDR 597
LCDTR 715
LCEBR 655
LCER 597
LCGFR 194
LCGR 192
LCR 187
LCXBR 655
LCXR 597

machine instruction *(continued)*
LD 568
LDE 619
LDEB 665
LDEBR 665
LDER 619
LDETR 716
LDGR 570, 700
LDR 568, 700
LDXBR 664
LDXR 616
LDXTR 717
LDY 568
LE 568
LEDBR 664
LEDR 616
LEDTR 717
LER 568, 700
LEXBR 664
LEXR 616
LEY 568
LFAS 651
LFPC 651
LG 189
LGB 196
LGBR 196
LGDR 570, 700
LGF 194
LGFI 319
LGFR 194
LGH 189
LGHI 319
LGHR 192
LGR 192
LH 182
LHI 319
LHR 192
LLC 196
LLCR 196
LLGC 196
LLGCR 196
LLGF 196
LLGFR 196
LLGH 196
LLGHR 196
LLGT 196, 863
LLGTR 196, 863
LLH 196
LLHR 196
LLIHF 319
LLIHH 319
LLIHL 319
LLILF 319
LLILH 319
LLILL 319
LM 180
LMG 189
LMH 189
LNDBR 655
LNDFR 568
LNDR 597
LNDTR 715
LNEBR 655
LNER 597
LNGFR 194

machine instruction *(continued)*
LNGR 192
LNR 187
LNXBR 655
LNXR 597
LPDBR 655
LPDFR 568
LPDR 597
LPDTR 715
LPEBR 655
LPER 597
LPGFR 194
LPGR 192
LPR 187
LPXBR 655
LPXR 597
LR 187
LRDR 616
LRER 616
LRV 453
LRVG 453
LRVGR 453
LRVH 453
LRVR 453
LT 194
LTDBR 655
LTDR 597
LTDTR 715
LTEBR 655
LTER 597
LTG 194
LTGFR 194
LTGR 192
LTR 187
LTXBR 655
LTXR 597
LXD 619
LXDB 665
LXDBR 665
LXDR 619
LXDTR 715, 716
LXE 619
LXEB 665
LXEBR 665
LXER 619
LXR 568, 700
LZDR 569
LZER 569
LZXR 569
M 264, 265
MAD 627
MADB 672
MADBR 672
MADR 627
MAE 627
MAEB 672
MAEBR 672
MAER 627
MDB 657
MDBR 657
MDE 599
MDEB 657
MDEBR 657
MDER 599
MDR 599

machine instruction *(continued)*
MDTR 701
MDTRA 701
ME 599
MEE 599
MEEB 657
MEEBR 657
MEER 599
MER 599
MGHI 322
MH 264, 268
MHI 322
ML 264, 270
MLG 264, 270
MLGR 264, 270
MLR 264, 270
MP 497, 506
MR 264, 265
MS 264, 268
MSD 627
MSDB 672
MSDBR 672
MSDR 627
MSE 627
MSEB 672
MSEBR 672
MSER 627
MSG 264, 268
MSGF 264, 268
MSGFR 264, 268
MSGR 264, 268
MSR 264, 268
MSY 264, 268
MVC 365, 372
MVCIN 365, 373
MVCL 404, 405
MVCLE 404, 411
MVCLU 441, 442
MVCOS 374
MVI 353
MVIY 353
MVN 460
MVO 497, 516
MVST 415, 417
MVZ 460
MXBR 657
MXD 599
MXDB 657
MXDBR 657
MXDR 599
MXR 599
MXTR 701
MXTRA 701
N 288
NC 365, 376
NG 288
NGR 288
NI 353
NIHF 323
NIHH 323
NIHL 323
NILF 323
NILH 323
NILL 323
NIY 353

machine instruction *(continued)*
NR 288
O 288
OC 365, 376
OG 288
OGR 288
OI 353
OIHF 324
OIHH 324
OIHL 324
OILF 324
OILH 324
OILL 324
OIY 353
operand formats 102
operands 96
OR 288
PACK 460, 471
PC 950
PFPO 745
PKA 460, 478
PKU 460, 478
QADTR 722
QAXTR 722
RLL 242
RLLG 242
RR-type 107
RRDTR 724
RRXTR 724
RS-type 111
RX-type 108
S 216
SAM24 858
SAM31 858
SAM64 858
SBRM 651
SD 606
SDB 661
SDBR 661
SDR 606
SDTR 701
SDTRA 701
SE 606
SEB 661
SEBR 661
SER 606
SFASR 651
SFPC 651
SG 216
SGF 230
SGFR 230
SGR 216
SH 216
SI-type 112
SL 224
SLA 242
SLAG 242
SLB 228
SLBG 228
SLBGR 228
SLBR 228
SLDA 242
SLDL 242
SLDT 720
SLFI 321

machine instruction *(continued)*
SLG 224
SLGF 230
SLGFI 321
SLGFR 230
SLGR 224
SLL 242
SLLG 242
SLR 224
SLXT 720
SP 497, 501
SPM 234
SQD 627
SQDB 671
SQDBR 671
SQDR 627
SQE 627
SQEB 671
SQEBR 671
SQER 627
SQXBR 671
SQXR 627
SR 216
SRA 242
SRAG 242
SRDA 242
SRDL 242
SRDT 720
SRL 242
SRLG 242
SRNMT 718
SRP 497, 511
SRST 415
SRSTU 441
SRXT 720
SS-type 113
SS-type length fields 113
ST 179
STC 184
STCM 185
STCMH 189
STD 568
STDY 568
STE 568
STEY 568
STFPC 651
STG 189
STH 182
STM 180
STMG 189
STMH 189
STRV 453
STRVG 453
STRVH 453
SVC 950
SXBR 661
SXR 606
SXTR 701
SXTRA 701
TAM 858
TCDB 654
TCEB 654
TCXB 654
TDCDT 693
TDCET 693

machine instruction *(continued)*
TDCXT 693
TDGDT 726
TDGET 726
TDGXT 726
TM 356
TMY 356
TP 497
TR 365, 379
TRE 421
TROO 444
TROT 444
TRT 365, 384
TRTE 450
TRTO 444
TRTR 365, 387
TRTRE 450
TRTT 444
UNPK 460, 474
UNPKA 460, 479
UNPKU 460, 479
X 288
XC 365, 376
XCGBR 666
XG 288
XGR 288
XI 353
XIHF 324
XILF 324
XIY 353
XR 288
ZAP 497, 499
machine instruction statement 74
machine language 73
definition 60, 1050
machine length 366
definition 115, 1050
macro argument 951
macro instruction
ABEND 956, 977
arguments 951
call 951
CLOSE 970
COND= argument 960
CONVERTI 82, 1016
CONVERTO 82, 1017
DCB 966
DCBD 970
DCBE 971
definition 83, 1050
DUMPOUT 82, 1018
ESPIE 973
ESTAE 977, 978
ESTAEX 977
execute form 952
FREEMAIN 957, 959, 961
FREEPOOL 970
GET 964
GETMAIN 957, 958, 961
IARV64 957
IHADCB 970
invocation 951
list form 952
MODE= argument 954
OPEN 969

macro instruction *(continued)*
PRINTLIN 82, 1018
PRINTOUT 82, 1019
PUT 966
R-form 954
READCARD 82, 1020
register arguments 954
RETURN 781
SAVE 777
SETRP 977
SPIE 973
STAE 977
standard form 952
STORAGE 957, 960, 961
SYSSTATE 955
system services 951
macro-instruction statement 74
MAD machine instruction 627
MADB machine instruction 672
MADBR machine instruction 672
MADR machine instruction 627
MAE machine instruction 627
MAEB machine instruction 672
MAEBR machine instruction 672
MAER machine instruction 627
mantissa
definition 572, 1050
mask bit
Floating-Point Control Register 649
Program Mask 234, 58
mask byte 356
mask digit 186, 191, 205
and execute instruction 361
branch 205, 330
MaxReal 582
definition 585, 1050
MDB machine instruction 657
MDBR machine instruction 657
MDE machine instruction 599
MDEB machine instruction 657
MDEBR machine instruction 657
MDER machine instruction 599
MDR machine instruction 599
MDTR machine instruction 701
MDTRA machine instruction 701
ME machine instruction 599
MEE machine instruction 599
MEEB machine instruction 657
MEEBR machine instruction 657
MEER machine instruction 599
memory address 43
alignment 44
memory dump
ABEND macro instruction 956
DUMPOUT macro instruction 957
MER machine instruction 599
message character 537
definition 549, 1050
MGHI machine instruction 322
MH machine instruction 264, 268
MHI machine instruction 322
millicode 48
definition 49, 1050
MinReal 582
definition 585, 1050
minuend
definition 238, 1050
ML machine instruction 264, 270
MLG machine instruction 264, 270
MLGR machine instruction 264, 270
MLR machine instruction 264, 270
mnemonic 82, 133
as abbreviation 106
definition 83, 115, 1050
extended 210
instruction name 82, 83, 133, 1050
modal instruction 308
modifier
bit length 259
byte length 259
definition 145, 1050
explicit length 140
exponent 138
length 138
scale 138
MP machine instruction 497, 506
MR machine instruction 264, 265
MS machine instruction 264, 268
MSD machine instruction 627
MSDB machine instruction 672
MSDBR machine instruction 672
MSDR machine instruction 627
MSE machine instruction 627
MSEB machine instruction 672
MSEBR machine instruction 672
MSER machine instruction 627
MSG machine instruction 264, 268
MSGF machine instruction 264, 268
MSGFR machine instruction 264, 268
MSGR machine instruction 264, 268
MSR machine instruction 264, 268
MSY machine instruction 264, 268
multiplicand
definition 284, 1050
multiplication
binary floating-point 657
decimal floating-point 702
fixed-point binary 264
process 272
fixed-point binary double-length product 265
fixed-point binary single-length product 267
floating-point 573
hexadecimal floating-point 599
logical 270
packed decimal 506
packed decimal operand order dependence 489
register-immediate 322
signed 265
multiplier
definition 284, 1050
multiply and add/subtract
binary floating-point 672
multiply and add/subtract *(continued)*
hexadecimal floating-point 627
multiply-defined symbols 123
MVC machine instruction 365, 372
MVCIN machine instruction 365, 373
MVCL machine instruction 404, 405
MVCLE machine instruction 404, 411
MVCLU machine instruction 441, 442
MVCOS machine instruction 374
MVI machine instruction 353
MVIY machine instruction 353
MVN machine instruction 460
MVO machine instruction 497, 516
MVST machine instruction 415, 417
MVZ machine instruction 460
MXBR machine instruction 657
MXD machine instruction 599
MXDB machine instruction 657
MXDBR machine instruction 657
MXDR machine instruction 599
MXR machine instruction 599
MXTR machine instruction 701
MXTRA machine instruction 701

N

N machine instruction 288
NaN (Not a Number)
binary floating-point 639
decimal floating-point 687
NC machine instruction 365, 376
NG machine instruction 288
NGR machine instruction 288
NI machine instruction 353
NIHF machine instruction 323
NIHH machine instruction 323
NIHL machine instruction 323
NILF machine instruction 323
NILH machine instruction 323
NILL machine instruction 323
NIY machine instruction 353
no-operation instruction 206
conditional 207
definition 214, 1051
NOALIGN assembler option 139
nominal value 138
definition 145, 1051
fixed-point binary
type F 146
type FD 146
type H 146
type HD 146
type of constant 138
non-modal instruction 309
non-overflowed zero
definition 496, 1051
NOP extended mnemonic 206, 211
NOPR extended mnemonic 206, 211
normalization 573
definition 585, 1051
notation
binary 17

notation *(continued)*
blank representation in examples 13
data fields 12
hexadecimal 18
instruction components 13
positional 16
space representation in examples 13
subscripts 12
NOTHREAD assembler option 810
NR machine instruction 288
null byte
definition 1051
null-terminated string 415
numeric digit 460
definition 483, 1051
in packed decimal 460

O

O machine instruction 288
OBJ object module
definition 867, 1051
object code 73
creation 73
definition 83, 1051
pass 2 generation 125
object module 831
creation 73
definition 83, 867, 1051
END record 832
ESD record 831
ESDID 831
RLD record 831
SYM record 831
TXT record 831
OC machine instruction 365, 376
offset
by ORG instruction 168
CUSE instruction 423
definition 530, 1051
during relocation 855
in Q-type constant 832
MVO instruction 516
with dependent USING 887
OG machine instruction 288
OGR machine instruction 288
OI machine instruction 353
OIHF machine instruction 324
OIHH machine instruction 324
OIHL machine instruction 324
OILF machine instruction 324
OILH machine instruction 324
OILL machine instruction 324
OIY machine instruction 353
ones' complement representation
definition 39, 1051
opcode
See also operation code
definition 115, 1051
OPEN macro instruction 969
operand 44
address 53
assembly-time 107
operand *(continued)*
definition 83, 1051
execution-time 107
halfword 182
immediate 112, 317
in an instruction description 13
in an instruction statement 13
machine instruction 44
optional 443
subject of an operation 13
operand order dependence 486, 489
definition 530, 1051
operation code
and instruction length 53
definition 60, 115, 1051
examples 54
operator
− 15, 97
/ 15, 97
× 15
* 15, 97
+ 15, 97
÷ 15
definition 15, 104, 1051
unary − 97
unary + 97
option
See also assembler option
*PROCESS statement 1051
ACONTROL assembler instruction 1051
definition 1051
GOFF 809, 849, 854
OR machine instruction 288
OR operation 289, 298, 1051
definition 298, 1051
register-immediate 323
register-register 291
storage-immediate 354
storage-storage 376
XOR operation
definition 298
order dependence
definition 496, 1051
ordinary control section
See also control section
common control section 1044
CSECT 1044
offsets fixed at assembly time 1044
positions at execution time 1044
relocation at later times 1044
RSECT 1044
ordinary symbol 89, 1048
external 89
internal 89
ordinary USING 120
definition 906, 1051
shortcomings 876, 881
ORG assembler instruction 167
extended syntax 168, 1051
definition 175, 1051
origin 80
definition 83, 1051
initial location 80
overflow
binary floating-point 649
binary floating-point rounding 664
definition 39, 1051
fixed-point binary 28, 31
hexadecimal floating-point 602
hexadecimal floating-point rounding 617
packed decimal 485
two's complement 31
overflowed zero
definition 496, 1051
overlay 847

P

PACK machine instruction 460, 471
packed decimal
addition process 492
addition rules 485
arithmetic rules 484
comparison 503
comparison rules 487
constant 467
convert to/from decimal floating-point 709
data exception 485
divide exception 491
division 509
division rules 490
editing overview 536
mixed integer-fraction representation 553
move with offset 516
multiplication 506
multiplication data exception 489
multiplication rules 489
operand order dependence 486, 489, 530, 1051
overflow 485, 486
representation 465
scale attribute 467
scaled arithmetic 522
general rules 522
shifting 511
sign-magnitude representation 466
specification exception 489, 491
subtraction process 493
subtraction rules 485
padding 152
by compare and move long 403
character 403
definition 158, 1051
hexadecimal floating-point constant 592
in constants 152
with blanks 153
with sign bits 153
with zeros 467
parameter 765
definition 788, 1051
parameterization 169
definition 175, 1052

partial-completion instruction 403
Partitioned Data Set Extended 849
pattern 536
character 537
definition 549, 1052
digit selector 537, 1052
digit selector and significance start 1052
digit selector and significance starter 537
editing 536
field separator 537, 1052
fill character 537, 1052
message character 537, 1052
payload
definition 738, 1052
PC machine instruction 950
PDSE 849
percolate 977, 980
PFPO machine instruction 745
PICA
See also Program Interruption Control Area
maskable program interruptions 974
PIE
See Program Interruption Element
pipeline 207
definition 214, 1052
PKA machine instruction 460, 478
PKU machine instruction 460, 478
PM
See Program Mask
positional argument
definition 981
post-normalization
definition 585, 1052
hexadecimal floating-point division 603
hexadecimal floating-point multiplication 599
postfix notation 923
definition 945, 1052
postorder tree traversal 940
definition 945, 1052
pre-normalization
definition 585, 1052
hexadecimal floating-point division 603
hexadecimal floating-point multiplication 599
precision
definition 530, 1052
preferred exponent 698
definition 738, 1052
preferred quantum 697
definition 738, 1052
preferred sign codes
zoned and packed decimal numbers
definition 483, 1052
preorder tree traversal 940
definition 945, 1052
PRINTLIN macro instruction 1018
definition 1026
PRINTLIN macro instruction *(continued)*
description 1018
PRINTOUT macro instruction 1019
definition 1026
description 1019
Private Code section 826
privileged operation 48
privileged operation exception 57
problem state 48
definition 49, 1052
process
binary division 280
binary multiplication 272
hexadecimal floating-point addition 612
hexadecimal floating-point complement addition 613
hexadecimal floating-point subtraction 612
packed decimal addition 492
packed decimal subtraction 493
program interruption 56
See also interruption
access exception 57
addressing 57
data 57
decimal divide 57
decimal overflow 57
definition 981, 1052
ESPIE macro 973
identifying token 973
execute exception 57
fixed-point divide 57
fixed-point overflow 57
hexadecimal floating-point divide 57
hexadecimal floating-point exponent overflow 57
hexadecimal floating-point exponent underflow 57
hexadecimal floating-point lost significance 57
invalid operation 57
PIE 973
privileged operation 57
Program Mask control
decimal overflow 58
fixed-point overflow 58
hexadecimal floating-point exponent underflow 58
hexadecimal floating-point lost significance 58
specification 57
SPIE macro 973
Program Interruption Control Area 973
Program Interruption Element 975
contents 975
DSECT EPIE 975
mapping macro IHAEPIE 975
Program Interruption Exit 975
program length
definition 115, 1052
program linking 833
program linking *(continued)*
combining object modules 833
Composite External Symbol Dictionary 835
definition 867, 1052
load modules 845
overview 73
Program Loader 73, 116, 809
boundary alignment 139, 809
definition 83
load modules 856
relocation 73
program loading
boundary alignment 809
Program Loader 73
Program Mask 47, 234, 602, 607
decimal overflow 58
definition 60, 1052
fixed-point overflow 58
hexadecimal floating-point exponent underflow 58, 602
hexadecimal floating-point lost significance 58, 607
mask bits 234
retrieve/set 234
program object 848
definition 867, 1052
Partitioned Data Set Extended 849
PDSE 849
program object class 849
program objects
Program Status Word 47
basic addressing mode 308
Condition Code 47, 204
definition 49, 1052
extended addressing mode 308
Instruction Address 47, 50
instruction address vs. location counter 92
Instruction Length Code 47
interruptions 55
new 56
old 56
Program Mask 47
switching 55
pseudoregister 1046
definition 867, 1052
PSW
See Program Status Word
PUT macro instruction 966

Q

Q-type address constant
definition 867, 1052
QADTR machine instruction 722
QAXTR machine instruction 722
quadword 44, 177
qualified symbol 882
definition 906, 1052
qualifier 882
definition 906, 1052
quantum 683
decimal floating-point quantize 722

quantum *(continued)*
definition 738, 1052
preferred 696, 697
quantum exception
decimal floating-point 698
queue 934
definition 945, 1052
quotient
definition 284, 1052
fixed-point binary division 274
packed decimal 490

R

radix 560
definition 572, 1053
radix point 553
radix-complement representation 24
definition 39, 1053
READCARD macro instruction 1020, 1020
definition 1028
description 1020
real address 67
address translation 67
DAT 67
definition 69, 1053
real numbers 745
realistic numbers 745
recipe
binary overflow detection 31
binary subtraction 35
two's complementation 27
recovery routine 977
definition 981
recovery/termination manager 977
percolate 977
retry routine 977
RTM 977
recursion 987
definition 992
reenterability 983, 984
assembly time 984
definition 992
linking time 984
RSECT 1053
reenterable
definition 1053
reentrant
See also reenterability
definition 993, 1053
reference control section 803
COM 803
definition 867, 1053
DSECT 804
DXD 804
reference table
ASCII representation 1013
DC statement types 1014
EBCDIC representation 1012
fractions in hexadecimal and decimal 1011
hexadecimal addition 996
hexadecimal digits 995
hexadecimal multiplication 996
reference table *(continued)*
integers in hexadecimal and decimal 1003
powers of 10 in hexadecimal 1001
powers of 16 1000
powers of 2 997
register
access 48
control 48
floating-point register 45, 564
floating-point register pair 565
general register 45
general register pair 242, 274
relative-immediate 305
address generation 305
relocatability
complex 1044
relocatability attribute 132
complex 132
relocatable 91, 118
absolute 104, 1041
complex 104, 1044
definition 95, 1053
location counter reference 97
simple 104, 1054
relocatable program 126
relocate
definition 867, 1053
relocating loader
definition 867, 1053
relocation 73, 92
by Program Loader 855
definition 83, 95, 867, 1053
load module creation 855
of A-type constant 837
of V-type constant 837
relocation attribute 90
complex 807
definition 825
dummy control section 873
ESDID 818
in expression evaluation 98
in External Symbol Dictionary 93
relocation dictionary
definition 868, 1053
remainder
binary floating-point 668
definition 284, 1053
fixed-point binary division 274
hexadecimal floating-point 625
in shift instructions 248
packed decimal 490
RENT assembler option 984
representation 639, 641
arithmetic vs. logical 37
ASCII 432, 1013
BCD 429
Big-Endian 1043
binary floating-point 639, 641
binary numbers 24
blanks in examples 13
decimal floating-point 686
decimal floating-point (conceptual) 682
representation *(continued)*
decimal floating-point encoding 684
diminished radix-complement 24
EBCDIC 87, 430, 1012
hexadecimal 18
hexadecimal floating-point 586
Little-Endian 1049
logical 25
packed decimal 465
radix-complement 24
redundant decimal floating-point 683
sign extension 30
sign-magnitude 24, 466
spaces in examples 13
Unicode 438
zoned decimal 460
residence mode
RMODE 827
RMODE assembler instruction 827
retry 980
retry routine 977
return address 765
definition 788, 1053
return code 779
definition 788, 1053
return flag 778
RETURN macro instruction 781
returned values 762
RLD record in object module 831
A-type constant 831
Cumulative External Dummy 832
Q-type constant 832
V-type constant 832
RLL machine instruction 242
RLLG machine instruction 242
RMODE
definition 868, 1053
rotating shift 257
definition 261, 1053
rounding
binary floating-point 646
decimal floating-point 695
decimal floating-point reround 724
rounding digit 575
definition 585, 1053
rounding instructions
binary floating-point 664
decimal floating-point 717
hexadecimal floating-point 616
packed decimal 511
rounding mode
binary floating-point 650
decimal floating-point 718
rounding-mode suffix 595, 643
binary floating-point constants 643
decimal floating-point constants 691
hexadecimal floating-point constants 595
row order 910

row order (continued)
definition 945, 1054
row-major order
definition 945, 1054
RR-type machine instruction 107
RRDTR machine instruction 724
RRXTR machine instruction 724
RS-type machine instruction 111
RSECT
control section 1054
definition 1054
External Symbol Dictionary 1054
reenterable 1054
RSECT assembler instruction 803, 984
RTM 977
run time
See also execution time
definition 1054
RX-type machine instruction 108

S

S machine instruction 216
S-type address constant 149
S-type constant 149
SAM24 machine instruction 858
SAM31 machine instruction 858
SAM64 machine instruction 858
save area
32-bit registers 770
chaining 772
doubly-linked list 771
extended conventions 773
standard 18-word 770
SAVE macro instruction 777
SBCS 434
SBRM machine instruction 651
scale modifier
fixed-point binary constant 555
hexadecimal floating-point constant 592
scaled arithmetic
definition 530, 1054
SD machine instruction 606
SDB machine instruction 661
SDBR machine instruction 661
SDR machine instruction 606
SDTR machine instruction 701
SDTRA machine instruction 701
SDWA 977
SE machine instruction 606
SEB machine instruction 661
SEBR machine instruction 661
SECTALGN assembler option 139, 156, 809
section
See also control section
alignment 809
common 803
control 803
definition 868, 1054
dummy 804
entry point 818
external dummy 804
section (continued)
linking 833
literals 808
private code 826
program object section 849
resuming 807
types 803
segment 68
definition 868, 1054
self-defining term
assembly-time constants 85
binary 85
character 86
ampersand 86
apostrophe 86
translation 88, 433
DBCS 436
decimal 85
definition 95, 1054
hexadecimal 85
no embedded blanks 86
vs. C-type constant 150
sequence symbol 89
SER machine instruction 606
set rounding mode
binary floating-point 651
decimal floating-point 718
SETRP macro instruction 977
sexadecimal representation 18
SFASR machine instruction 651
SFPC machine instruction 651
SG machine instruction 216
SGF machine instruction 230
SGFR machine instruction 230
SGR machine instruction 216
SH machine instruction 216
shift
bit bucket 244
decimal floating-point significand 720
double-length arithmetic 252
double-length logical 248
general register pair 242
packed decimal 511
process 243
rotating 257
single-length arithmetic 252
single-length logical 245
unit 243
Shift-In
definition 1054
shift-in character 434
Shift-Out
definition 1054
shift-out character 434
SI-type instructions 352
SI-type machine instruction 112
sign extension 30, 183
definition 203, 1054
signed 20-bit displacement 303
sign-magnitude representation 24
definition 39, 1054
packed decimal 466
significance indicator 537
definition 549, 1054
significance starter 537
definition 549, 1054
significand
binary floating-point 639
decimal floating-point 687
definition 572, 1054
hexadecimal floating-point 560
simple I/O macros
$$GENIO 1023
CONVERTI 1016
CONVERTO 1017
DUMPOUT 1018
installation 1023
PRINTLIN 1018
PRINTOUT 1019
READCARD 1020
usage notes 1021
Single-Byte Character Set
definition 1054
SIY-type instructions 352
SL machine instruction 224
SLA machine instruction 242
SLAG machine instruction 242
SLB machine instruction 228
SLBG machine instruction 228
SLBGR machine instruction 228
SLBR machine instruction 228
SLDA machine instruction 242
SLDL machine instruction 242
SLDT machine instruction 720
SLFI machine instruction 321
SLG machine instruction 224
SLGF machine instruction 230
SLGFI machine instruction 321
SLGFR machine instruction 230
SLGR machine instruction 224
SLL machine instruction 242
SLLG machine instruction 242
SLR machine instruction 224
SLXT machine instruction 720
SP machine instruction 497, 501
space
See also blank
ASCII representation 433
EBCDIC representation 88
representation in examples 13
text representation 15, 1054
specification exception 57
packed decimal 489
SPIE macro instruction 973
program interruption exit 973
SPM machine instruction 234
SQD machine instruction 627
SQDB machine instruction 671
SQDBR machine instruction 671
SQDR machine instruction 627
SQE machine instruction 627
SQEB machine instruction 671
SQEBR machine instruction 671
SQER machine instruction 627
square root
binary floating-point 671
hexadecimal floating-point 626
hexadecimal floating-point interruption 627

SQXBR machine instruction 671
SQXR machine instruction 627
SR machine instruction 216
SRA machine instruction 242
SRAG machine instruction 242
SRDA machine instruction 242
SRDL machine instruction 242
SRDT machine instruction 720
SRL machine instruction 242
SRLG machine instruction 242
SRNMT machine instruction 718
SRP machine instruction 497, 511
SRST machine instruction 415
SRSTU machine instruction 441
SRXT machine instruction 720
SS-type machine instruction 113
ST machine instruction 179
stack 923
  definition 945, 1055
  pop 924
  push 924
STAE macro instruction 977
START assembler instruction 803
statement 73
  *PROCESS statement 1051
  ACONTROL assembler instruction 1051
  column positions
    begin column 75
    continue column 75
    end column 75
  CSECT 80
  DC 137
  definition 83, 1055
  dependent USING 885
  DROP assembler instruction 128
  DS 159
  END 80
  EQU 93
  EQU assembler instruction 162
  free-field 77
  job control 81
  labeled dependent USING 891
  labeled USING 882
  LOCTR assembler instruction 810
  LTORG 156
  ORG 167
  ORG extended syntax 1051
  RSECT 1054
  START 80
  USING 126
statement field
  comment 76
  definition 84, 1055
  name 76
  operand 76
  operation 76
  remarks 76
statement type
  Assembler 74
  comment 74
  machine instruction 74
  macro-instruction 74
status preservation 763
STC machine instruction 184
STCM machine instruction 185
STCMH machine instruction 189
STD machine instruction 568
STDY machine instruction 568
STE machine instruction 568
STEY machine instruction 568
STFPC machine instruction 651
STG machine instruction 189
STH machine instruction 182
STM machine instruction 180
STMG machine instruction 189
STMH machine instruction 189
STORAGE macro instruction 957, 960, 961
  subpool 961
store operation 179
  definition 203, 1055
STRV machine instruction 453
STRVG machine instruction 453
STRVH machine instruction 453
subpool 961
  FREEMAIN 961
  GETMAIN 961
  STORAGE 961
subroutine 756
  argument 765
  argument passing 759, 766
    32-bit addresses 766
  entry point 765
  internal 798
  linkage 757
  linkage convention 765
  lowest level 785
  parameter 765
  return address 765
  returned values 762
  status preservation 763
  without addressability 797
subtraction
  binary floating-point 661
  binary integer 32
  decimal floating-point 703
  fixed-point binary 216
  floating-point 578
  hexadecimal floating-point 606
  logical 224
  packed decimal 485
  packed decimal operand order dependence 486
  packed decimal process 493
  register-immediate 321
  with borrow 228
subtrahend
  definition 238, 1055
supervisor state 48
  definition 49, 1055
SVC machine instruction 950
SXBR machine instruction 661
SXR machine instruction 606
SXTR machine instruction 701
SXTRA machine instruction 701
SYM record in object module 831
symbol
  absolute 90
symbol *(continued)*
  attribute 95, 1042, 1055
    assembler 166
    definition 95
    integer 90
    length 90, 143
    program 166
    relocation 90, 125
    scale 90
    type 90
    value 90, 125
  complexly relocatable 99
  defined 90
  definition 95, 1055
  duplicate 123
  external 89, 818
  external symbol 1046
  internal 89, 91
  internal symbol 1048
  multiply defined 123
  ordinary 89
    external 89
    internal 89
  ordinary symbol 1048
  qualified symbol 882
  qualifier 882
  relocatable 90
  sequence 89
  undefined 123
  value vs. variable 94
  variable 89
symbol attribute reference
  definition 104, 1055
  integer 96
  length 96
  scale 96
Symbol Table 123
  definition 133, 1055
  External Symbol Dictionary 93
  external symbol table 89
  ordinary symbols 123
symbolic operand
  binary floating-point (DMin) 644
  binary floating-point (Inf) 644
  binary floating-point (Max) 644
  binary floating-point (Min) 644
  binary floating-point (NaN) 644
  binary floating-point (QNaN) 644
  binary floating-point (SNaN) 644
  decimal floating-point (DMin) 690
  decimal floating-point (Inf) 690
  decimal floating-point (Max) 690
  decimal floating-point (Min) 690
  decimal floating-point (NaN) 690
  decimal floating-point (QNaN) 690
  decimal floating-point (SNaN) 690
  hexadecimal floating-point symbolic operand (DMin) 595
  hexadecimal floating-point symbolic operand (MAX) 595
  hexadecimal floating-point symbolic operand (Min) 595

Syntactic Character Set
  definition 1055
SYSSTATE macro instruction 955
System Diagnostic Work Area 977
system interruption
  definition 982
system service
  definition 982

T

table 914
  definition 945, 1055
TAM machine instruction 858
target instruction
  definition 398
TCDB machine instruction 654
TCEB machine instruction 654
TCXB machine instruction 654
TDCDT machine instruction 693
TDCET machine instruction 693
TDCXT machine instruction 693
TDGDT machine instruction 726
TDGET machine instruction 726
TDGXT machine instruction 726
term
  definition 104, 1055
  illustration 104
  literal 96
  location counter reference 96
  self-defining 96
  symbol 96
  symbol attribute reference
    integer 96
    length 96
    scale 96
TEST assembler option 89
text 831
  definition 868, 1055
  in load module 846
  machine language 831
threading
  Location Counter values 809
time 1054
  assembly 74, 1043
  binding 1043
  execution 74, 1043
  linking 1043
  run time
    See execution time
TITLE assembler instruction 78
TM machine instruction 356
TMY machine instruction 356
TP machine instruction 497
TR machine instruction 365, 379
trailing significand field 686
transformation format
  (Unicode) 438
TRANSLATE assembler option 88, 158, 433
translate table 379, 384, 387
TRE machine instruction 421
tree
  B-tree 940
  binary 937
tree search
  inorder traversal 940
  postorder traversal 940
  preorder traversal 940
TROO machine instruction 444
TROT machine instruction 444
TRT machine instruction 365, 384
TRTE machine instruction 450
TRTO machine instruction 444
TRTR machine instruction 365, 387
TRTRE machine instruction 450
TRTT machine instruction 444
true addition
  definition 1055
true decimal addition
  definition 496, 1055
truncation 152
  definition 158, 1055
  hexadecimal floating-point
    constant 592
  in constants 152
two's complement overflow 31
two's complement representation
  definition 39, 1055
  recipe 27
TXT record in object module 831
type extension 157
  all types 1014
  definition 158, 1055
  type AD 157
  type CA 157
  type CE 157
  type CU 157
  type DB 567, 642
  type DD 567
  type DH 567
  type EB 642
  type EH 567
  type FD 157
  type LB 567, 642
  type LD 567
  type LH 567
  type LQ 594

U

U-format records 967
ulp (unit in the last place) 581
  definition 585, 1055
  relative precision 581
unbiased rounding
  definition 530, 1055
undefined length records 967, 969
undefined symbol 123
underflow
  binary floating-point 649
  hexadecimal floating-point 602
Unicode
  C-type constant 151
  constant 439
  CU-type constant 157
  definition 1055
  glyph 439
  numeric character 462, 478
    definition 483, 1055
Unicode *(continued)*
  pack 478
  transformation format 447
  translate instructions 444
  unpack 479
Unicode numeric character
  definition 1055
unit shift 243
unnormalized addition
  hexadecimal floating-point 609
unnormalized hexadecimal floating-point representation 588
unnormalized subtraction
  hexadecimal floating-point 609
UNPK machine instruction 460, 474
UNPKA machine instruction 460, 479
UNPKU machine instruction 460, 479
USING assembler instruction 120, 126, 882, 885, 891
  absolute USING location 131
  base location 120, 126
  base register 120
  definition 133, 1056
  dependent 1043, 1045
  dependent USING 885
  labeled 882, 1049
  labeled dependent 891, 1045
  ordinary 120, 882
  qualified symbol 1049
USING Map 133
USING Table 125, 128, 133
  definition 134, 1056
  USING Map 133
USING(MAP) option 133
UTF-16 447
UTF-32 447
UTF-8 447

V

V-format records 967
V-type address constant 820, 868, 1056
  definition 868, 1056
variable
  vs. symbol value 94
variable length records 967, 968
variable symbol 89
variable symbols
  attribute 1042
    symbol itself 1042
    symbol's value 1042
variable-length argument list 767
virtual address 67
  address translation 67
  DAT 67
  definition 69, 1056
virtual origin 912
  definition 945, 1056
  of array 912
Von Neumann Architecture 5
Von Neumann, John 5
VSAM 971

W

ward in double-byte EBCDIC 435
widget 526, 528, 727
word 44, 177
workstation data 453
WXTRN assembler instruction 818, 820
  difference from EXTRN 820
WXTRN statement 1046

X

X machine instruction 288
XC machine instruction 365, 376
XCGBR machine instruction 666
XG machine instruction 288
XGR machine instruction 288
XI machine instruction 353
XIHF machine instruction 324
XILF machine instruction 324
XIY machine instruction 353
XOR operation 289, 298, 1056
  definition 298, 1056
  register-immediate 324
  register-register 292
  storage-immediate 354
  storage-storage 376
XR machine instruction 288

Z

z/Architecture 3
ZAP machine instruction 497, 499
zero
  unexpected floating-point behavior 747
zero duplication factor 160
  definition 175, 1056
zero extension 196
  definition 203, 1056
zone digit 460
  ASCII characters 462
  definition 483, 1056
  Unicode characters 462
zoned decimal
  constant 464
  numeric digit 460
  preferred sign 462
  representation 460
  sign digit 460
  zone digit 460
zoned digit 546
  definition 549, 1056
  in edited data 546