Assembler Language Programming

for

IBM System z™ Servers

Lecture Slides

Version 2.00, Chapters I to VIII

John R. Ehrman

IBM Silicon Valley Lab
ehrman@us.ibm.com

Note:

Slides are keyed in their bottom left corner to the text, referring to the related Chapter and Section.

Second Edition (March 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

After June 1, 2016, please address them to

john.ehrman@comcast.net

- The major divisions of the text are:

Chapter I: Number representations, arithmetic, and base conversions

Chapter II: CPU, registers, PSW, and instructions

Chapter III: Fundamentals of Assembler Language

Chapter IV: Defining data and work areas

Chapter V: Basic instructions using General Registers; branches

Chapter VI: Addressing, loop instructions, immediate operands

Chapter VII: Bit and character data, instructions, and representations

Chapter VIII: Packed decimal data and instructions

Chapter IX: Floating-point data and instructions

Chapter X: Large programs, modularization, and subroutines

Chapter XI: Dummy control sections and enhanced USING statements

Chapter XII: System services, exception handling, reenterability, recursion

Appendix A: Conversion and reference tables

Appendix B: Useful macros for conversion, reading, display and printing

- Programming Environments
  - We assume your programs will execute on one of IBM's z/OS (MVS), z/VM (CMS), or z/VSE operating systems
    - The simple macros in Appendix B have worked on all three
- Some section headings end with "**(*)**"
  - These may be more detailed or difficult topics
- The Exercises and Programming Problems are strongly recommended
  - Their numbers are followed by a parenthesized digit with an estimated difficulty from (1) = easy, to (5) = difficult
  - Recommended exercises and problems are tagged with "+"

- A computer can be viewed at many levels, such as
  - A collection of logical circuits
  - Techniques used to make circuits perform operations like addition, division
  - Instructions to perform an operation like addition, data movement
  - A processor to evaluate mathematical expressions, maintain data
  - A simulator of physical processes: traffic flow, weather prediction
- Our concern is mainly the middle level
  - With occasional excursions into neighboring levels
- The assembler described is IBM's "IBM High Level Assembler for z/OS & z/VM & z/VSE", known as "HLASM"

Why Program in Assembler Language (and Why Not)? 4

Why?

- You want to learn more about how a powerful processor works
- Assembler Language helps you understand what language compilers do
- You need functions not provided by your “high-level” language
- You want to use instructions not supported by your “high-level” language
- It’s more fun: you can write instructions the way you want
- It’s stable: you don’t need to retest everything when using an updated assembler
- It’s parameterizable: you can modify a small number of statements and reassemble to make substantial updates to your program
- The language is extensible using HLASM’s powerful macro facility
  - Unfortunately, that topic is beyond the scope of this text

Why not?

- Assembler Language can be verbose: more statements to do something
  - But the extreme brevity of some “modern” languages can be hard to learn
- The language can be *too* flexible for some
- Some instructions don’t follow simple, regular usage rules
- Programs can sometimes be harder to debug
- Lack of a run-time library
  - It’s usually easy to access modules in an existing library
- Lack of portability: Assembler Language programs are by definition targeted to IBM’s System z processor family

If you have good reasons to use other languages, by all means do!

Assembler Language Misconceptions 6

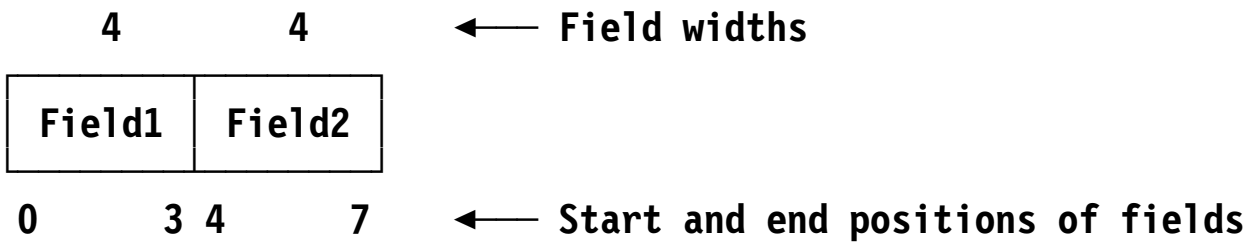
- It's dead!
  Many organization have major investments in Assembler Language applications that provide functionality and speed; frequent updates are a business necessity
- It's hard to learn!
  The *language* is very simple; difficulties can be due to programming style or instruction unfamiliarity
- It's hard to maintain!
  Research has shown there's little difference among languages in maintenance costs; "clean" (and messy) programs can be written in any language
- Converting to another language is easy!
  This is very rarely true, and attempts can be *very* expensive
- You can't do structured programming!
  The HLASM "Toolkit Feature" has a powerful set of Structured Programming Macros

Chapter I: Starting With System z Assembler Language 1

This chapter reviews some basic aspects of System z processors

- Section 1 introduces notation, terminology, and conventions
- Section 2 describes basic properties of the number representations used in System z processors:
  - Binary and hexadecimal numbers
  - Arithmetic and logical representations
  - 2's complement arithmetic
  - Conversions among number representations

- When we describe a “field” (an area of memory, part of a register) we often use a figure like this:



  We number positions from **left** to **right**.
- When we refer to a sequence of similar items, we may use subscripts like B_j, or appended letters like Bj, or the programming-language subscript notation B(j)
- The contents of some item X is often denoted c(X)
- The operators + − * / represent addition, subtraction, multiplication, and division, respectively
- To show a blank space, we sometimes use a • character

- The word "operand" is used in three senses:
  1. In the *z/Architecture* Principles of Operation (or "zPoP"), you may see a machine instruction described as

     LM R_1,R_3,D_2(B_2)

     where c(R_1) is the first operand, and a memory address is determined from D_2(B_2); but "operands" 1, 2, 3 are shown in order 1, 3, 2
  2. In Assembler Language, operands are defined by sequential position:

     ```
     LM    2,12,SaveArea
     ```

     the first operand is "2", the second is "12", and the third is "SaveArea".
  3. During execution, an operand is the subject of an instruction's operation:

     ```
     LM    2,12,SaveArea
     ```

     so c(GR2), c(GR12), and c(SaveArea) are all operands that are subjected to an operation.
- The intended meaning is usually clear from context

Section 2 describes fundamentals of number representations:

- Binary and hexadecimal numbers, and positional notation
- Conversion among different representations
- Logical (unsigned) and arithmetic (signed) representations
- Two's complement (signed) representation
- Binary addition and subtraction; overflow; signed vs. unsigned results
- Alternative representations of signed binary values

We'll start with integer values (no fractional parts):

- Decimal integer values like 1705 mean 1000+700+00+5, or $1\times10^3 + 7\times10^2 + 0\times10^1 + 5\times10^0$ (base 10)
- In binary, the number B'11010' means 10000 + 1000 + 000 + 10 + 0, or $1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$ (base 2)
  - So B'11010' = 16+8+2 = 26; B'1010' = 10, B'1111100111' = 999
  - Decimal numbers are written normally, binary numbers as B'nnn'
  - The term "binary digit" is usually abbreviated "bit"
- Exercise: convert to binary: 81, 255
- Exercise: convert to decimal: B'10101010', B'11110001'

Because the number of bits grows rapidly as numbers get larger, we use groups of 4 bits called “hexadecimal” or “hex” (base 16)

- The 16 possible hex values from 0 to 15 are represented by 0-9, A-F

0000	0001	0010	0011	0010	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	(bin)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(dec)
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	(hex)

- Hexadecimal numbers are written X′nnn′
  - So B′11010′ = X′1A′; B′1011′ = X′B′, B′1111100111′ = X′3E7′
- Exercise: convert to hexadecimal: 145, 500
- Exercise: convert to decimal: X′763′, X′F7′
- Exercise: convert to binary: X′763′, X′F7′

- Numbers like 2345 in some base A are written

  $2 \times A^3 + 3 \times A^2 + 4 \times A^1 + 5 \times A^0$

- If we write digits in order of decreasing significance as

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

  then a number X in base A is

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

- To convert X to a new base B, so that

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

  1. Divide X by B, save the quotient; the remainder is the low-order digit e_0
  2. Divide the quotient by B, save the quotient; the remainder is digit e_1
  3. Repeat until the quotient is zero
  4. The remainder digits are created in order of *increasing* significance

- Exercise: convert 2345 to bases 16, 7 and 13

- Three basic representations; first two used on System z:
  - Radix-complement (for System z's binary numbers: 2's complement)
  - Sign-magnitude (the way we write numbers: +5, −17)
  - Diminished radix-complement (no longer used in modern machines)
- Unsigned binary numbers ("logical" representation)
  - Every bit has *positive* weight
  - For an 8-bit integer, the most significant bit has weight $+2^7$
    - So (unsigned) B'10000001' = $+2^7 + 2^0 = 129$
- Signed binary numbers ("arithmetic" representation)
  - Every bit has positive weight, *but* the high-order bit has *negative* weight
  - For an 8-bit integer, the most significant bit has weight -2^7
    - So (signed) B'10000001' = $-2^7 + 2^0 = -127$
- Exercise: convert B'10101010' (signed and unsigned) to sign-magnitude decimal

- Binary addition is very simple:

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

- Finding the two's complement (negation) of a binary number:
  - Take its ones' complement: change all 0s to 1s and 1s to 0s; then add a low-order 1 bit
    - Examples, using signed 8-bit values:

```
 10000001  (signed −127)          00000001  (signed +1)
 01111110  ones' complement       11111110  ones' complement
+       1                       +       1
 01111111  (signed +127)          11111111  (signed −1)

 11111101  (signed −3)           00011111  (signed +31)
 00000010  ones' complement       11100000  ones' complement
+       1                       +       1
 00000011  (signed +3)            11100001  (signed −31)
```

  - Carries out of the leftmost bit: ignored for unsigned, important for signed
    - But most arithmetic instructions take note of carries

- Binary numbers can be lengthened to greater precision by sign extension
- If the sign bit is copied to the left, the value of the number is unchanged in the new, longer representation
  - Examples, using signed 16-bit values extended from 8 bits:

```
11111111 10000001  (signed −127)        00000000 00000001  (signed +1)
00000000 01111111  (signed +127)        11111111 11111111  (signed −1)
```

  - Many instructions do sign extension automatically

- **All** bits are added; high-order carries are lost (but noted)
  - Examples, using signed 4-bit values (range -8 ≤ value ≤ +7):

```
 1111 (−1)     0010 (+2)     0100 (+4)
+0001 (+1)    +0010 (+2)    +0100 (+4)
 0000 (+0)     0100 (+4)     1000 (−8, overflow)
```

  - Arithmetic addition: overflow possible only when adding like-signed operands.
    - Actions vary: signed overflow can be ignored, or cause an “interruption”

- Unsigned (logical) addition: carries are noted, no overflows
  - Examples, using unsigned 4-bit values (range 0 ≤ value ≤ 15):

```
 1111 (15)          0010 (2)           1100 (12)
+0001 (1)          +0010 (2)          +1001 (9)
 0000 (0, carry)    0100 (4, no carry)  0101 (5, carry)
```

- Conditional branch instructions (described in Section 15) can test for overflow (arithmetic addition or subtraction) and carries (logical addition or subtraction)

- Subtraction is slightly more complicated than addition...
  1. Form the ones′ complement of the second (subtrahend) operand
  2. Add the first (minuend) and complemented second operands *and* a low-order 1 bit (but in a *single* operation!)
  - Examples, using signed 4-bit values:

```
 −1−(+1)              +2−(+2)               3−5
 1111 (−1)            0010 (+2)             0011 (+3)
 1110 (+1,comp)       1101 (−2,comp)        1010 (+5,comp)
+   1                +   1                 +   1
 1110 (−2, carry)     0000 (+0, carry)      1110 (−2, no carry)
```

  - Arithmetic subtraction: overflows possible; logical subtraction: carries are noted
- Adding the first operand directly to the two′s complement of the second operand works *almost*, but *not* all the time!

- Why must we add all three items at once?

  Why not just add the first operand directly to the two's complement of the second? An example shows why:

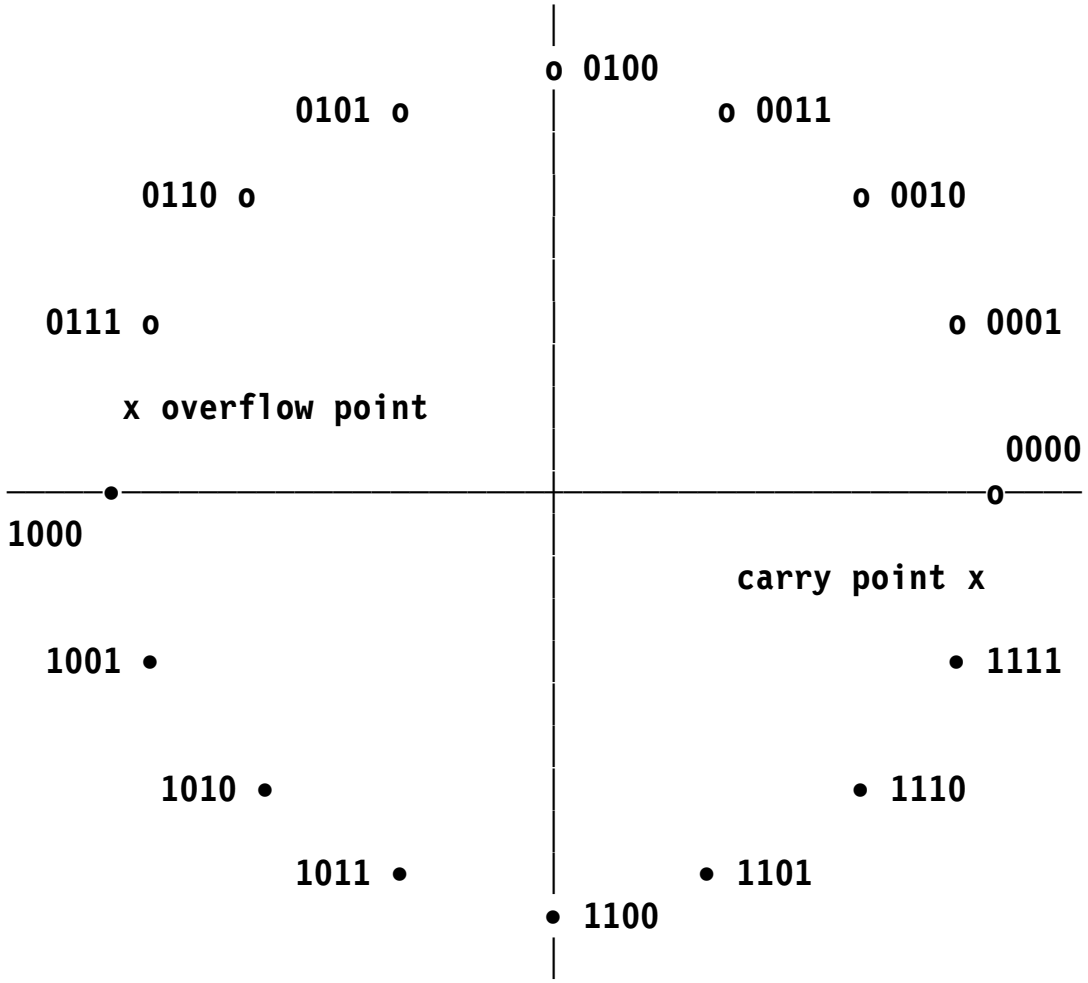
```
+1−(−8) right way          +1−(−8) wrong way
   0001 (+1)                  0001 (+1)
   0111 (−8, 1s comp)        +1000 (−8, 2's comp)
  +   1                       1001 (−7, no overflow)
   1001 (−7, overflow)
```

  - Overflow occurred in forming the two's complement of −8 *before* adding

- Adding all three items at once guarantees correct overflow detection

A Circular Representation of 4-bit Signed Integers 14

- A circular representation of 4-bit signed integers:



o = positive number,
• = negative number.

Addition: move counter-clockwise

Subtraction: move clockwise

Overflow: move past the overflow point

Carry from high-order bit: move past the carry point

- The bit patterns from logical and arithmetic add and subtract are identical; *only* the overflow or carry indications are different
- Other representations for binary numbers:

Binary Digits	*Logical Representation*	*Sign-Magnitude*	*Ones' Complement*	*Two's Complement*
0000	0	+0	+0	0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	−0	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

Exercise Answers

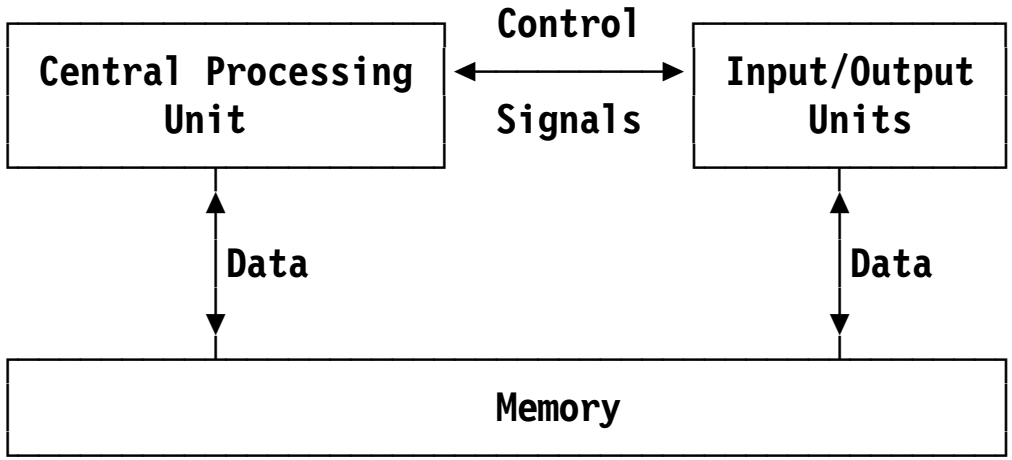
- Slide 5:
  - B′01010001′, B′11111111′
  - 170, 241
- Slide 6:
  - X′91′, X′1F4′
  - 1891, 247
  - B′11101100011′, B′11110111′
- Slide 7:
  - X′929′, 6560_7, $10B5_{13}$.
- Slide 8:
  - -86, 170

Chapter II. System z 1

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

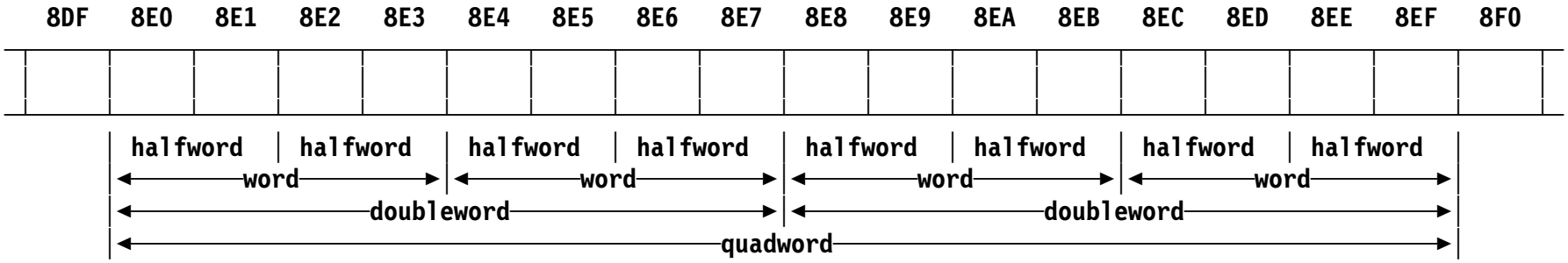
- Section 3 describes key processor structures: the Central Processing Unit (CPU), memory organization and addressing, general purpose registers, the Program Status Word (PSW), and other topics.
- Section 4 discusses the instruction cycle, basic machine instruction types and lengths, exceptions and interruptions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

- The three key elements of System z processors:



1. The CPU executes instructions, coordinates I/O and other activities
2. I/O units transfer data between Memory and external storage devices
3. Memory (“central storage”) holds instructions, data, and CPU-management data

- Memory storage unit is the 8-bit *byte*, each has its own address
  - Byte groups with addresses divisible by the group length have special names:

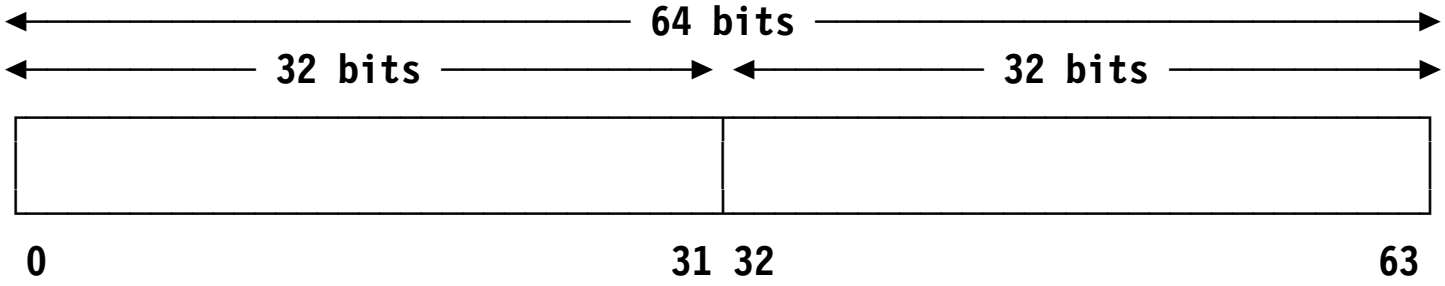


- Instructions may reference single bytes, groups as shown, or strings of bytes of (almost) any length

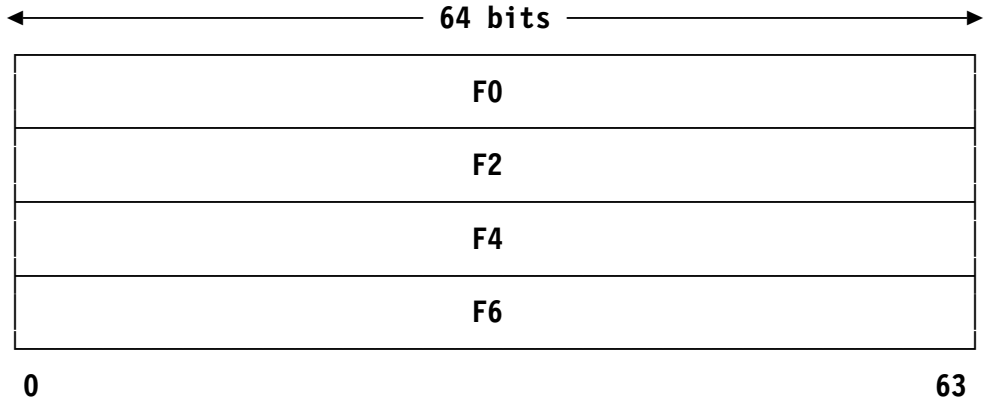
- Key elements are registers and the Program Status Word (PSW)
  - 16 “general purpose” registers, often arranged in even-odd-numbered pairs; used for arithmetic, logic, addressing

General Register 0	General Register 1
General Register 2	General Register 3
:	: :
General Register 12	General Register 13
General Register 14	General Register 15

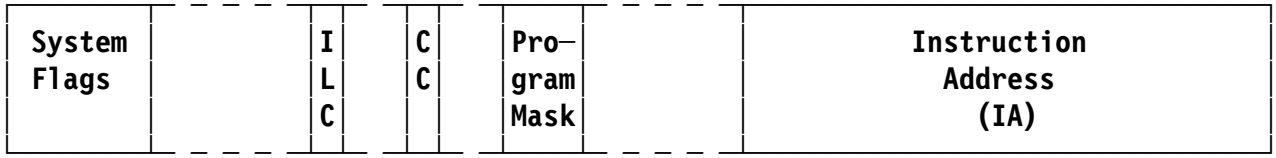
  - Each general register is usable as 64 bits or as two 32-bit halves



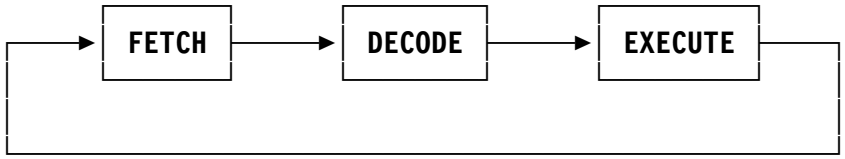
- Floating-point registers: now 16, originally 4:
  - Used for floating-point operations; some instructions use only the left half



- Program Status Word (PSW) (actually a 128-bit quadword)
  - Key components: Instruction Length Code (ILC), Condition Code (CC), Program Mask (PM), Instruction Address (IA)



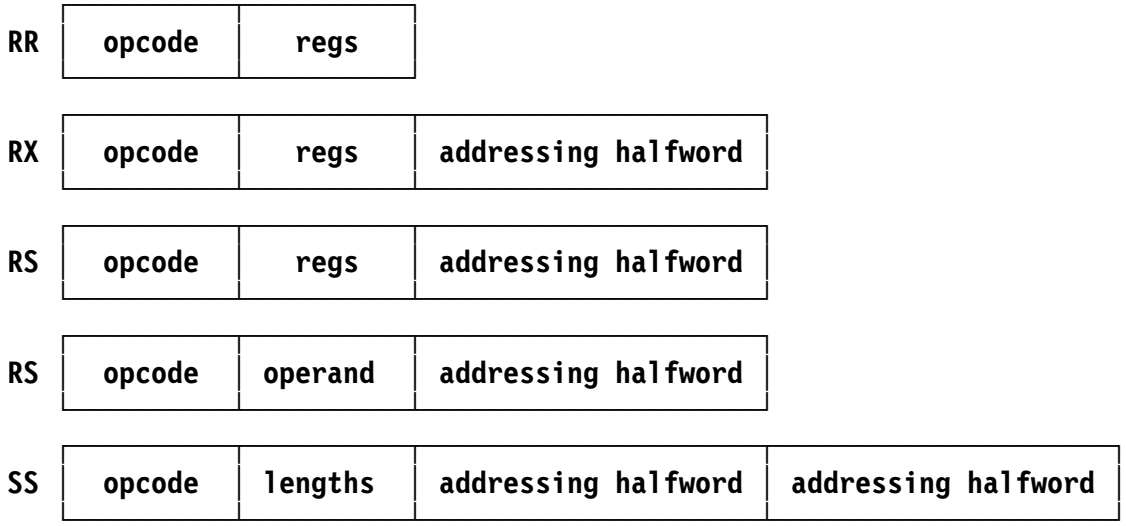
- Easiest to visualize in three steps:



1. Fetch: bring instruction from memory, determine its type and length
   - Add its length to PSW's Instruction Address (IA) to form address of "Next Sequential Instruction"
2. Decode: determine validity of instruction; access operands
3. Execute: perform the operation; update registers and/or memory as required

- Possible problems, interruptions (more at slides 9-10)
  - Fetch: invalid instruction address
  - Decode: invalid or privileged instruction
  - Execution: *many* possibilities!

- Original System/360 CPUs supported five instruction types:
  1. Register-Register (RR): operands entirely in registers
  2. Register-Indexed Storage (RX): Operands in registers and storage
  3. Register-Storage (RS): Operands in registers and storage
  4. Storage-Immediate (SI): Operand in memory *and* in the instruction
  5. Storage-Storage (SS): Operands in storage
- Instruction formats: 2, 4, or 6 bytes long



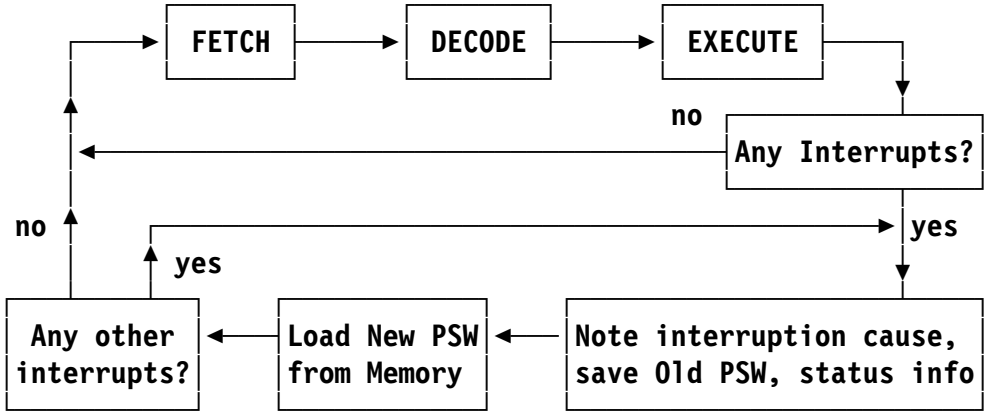
- *Every* instruction's first two bits of its first byte determine its length:

```
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
```

- Instruction Length Code (ILC) set to the number of *halfwords* in the instruction (1,2,3)

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

- The basic instruction cycle is modified to handle interruptions:



- There are six *classes* of interruption:
  1. Restart (operator action)
  2. External (timer, clock comparator)
  3. Machine Check (processor 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)

- The CPU saves the current ("old") PSW, loads a new PSW
  - Supervisor saves status information, processes the condition
  - Supervisor can return to interrupted program by loading old PSW
  - Applications are concerned almost entirely with Program interruptions
- Some "popular" Program Interruption Codes (IC):

IC=1 Invalid Operation Code.

IC=4 Access, Protection: program has referred to an area of memory to which access is not allowed.

IC=6 Specification Error: can be caused by many conditions.

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

IC=8 Fixed-Point Overflow: fixed-point binary result too large.

IC=9 Fixed-Point Divide Exception: quotient would be too big, or a divisor is zero.

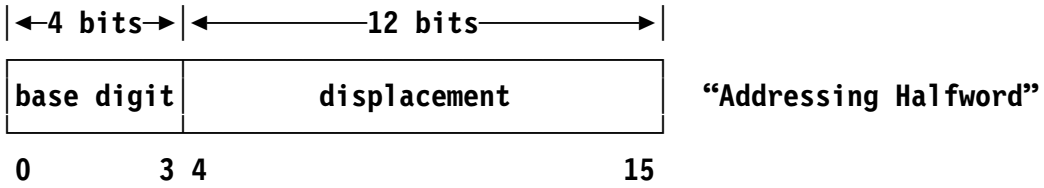
IC=A Decimal Overflow: packed decimal result too large.

IC=B Decimal Divide: packed decimal quotient too large, or a divisor is zero.

IC=C Hexadecimal floating-point exponent overflow: result too large.

IC=D Hexadecimal floating-point exponent underflow: result too small.

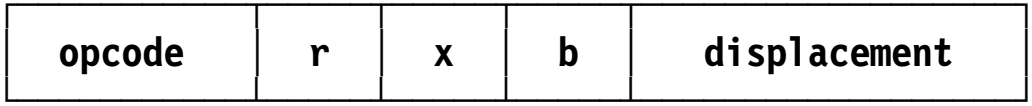
- System z instructions create many operand addresses using *base-displacement addressing*



  - The base register specification digit ("base digit") specifies one of general registers 1-15, the *base register* containing the *base address* or *base*
  - The displacement is an unsigned 12-bit integer
- Operand addresses:
  1. Copy displacement to internal Effective Address Register ("EAR")
  2a. If the base digit b is not zero, add c(Rb); ignore carries
  2b. If the base digit b is zero, do nothing
- The result in the EAR is the *Effective Address*.
  - Example with 32-bit values: an addressing halfword contains X'B2D5', and c(R11) = X'C73E90AF'. Then, in the EAR:

```
Step 1:    000002D5    displacement
Step 2a:  +C73E90AF    base
           C73E9384    Effective Address
```


- RX-type instructions contain an *index register specification digit* x:



  - Indexed Effective Address calculation adds two more steps:

3a. If the index digit x is not zero, add c(Rx); ignore carries
3b. If the index digit x is zero, do nothing

  - Example: RX-type instruction is X'431AB2D5', c(R10) = X'FEDCBA98', and c(R11) = X'C73E90AF'. In the EAR:

```
Step 1:    000002D5    displacement
Step 2a:   C73E90AF    base (from R11)
Step 3a:  +FEDCBA98    index (from R10)
         (1)C61B4E1C    Indexed Effective Address, carry ignored
```

- System z supports *Dynamic Address Translation* ("DAT")
  - DAT translates application addresses into "real" addresses
  - Helps Operating System better manage "actual" memory
    - Invisible to your program.

Chapter III. Assembler Language Programs 1

This chapter describes fundamental concepts of Assembler Language programming.

- Section 6 provides an overview of assembling, linking and loading for execution; conventions for preparing Assembler Language programs; and some helpful macro instructions that perform simple I/O and conversion operations.
- Section 7 discusses key concepts relating to symbols and “variables”.
- Section 8 investigates the elements of expression evaluation, and the basic Assembler Language operand formats used by instructions.
- Section 9 introduces typical instructions and how to write Assembler Language statements for them.
- Section 10 shows how the Assembler calculates displacements and assigns base register values in Addressing Halfwords, and introduces the important USING and DROP assembler instructions.

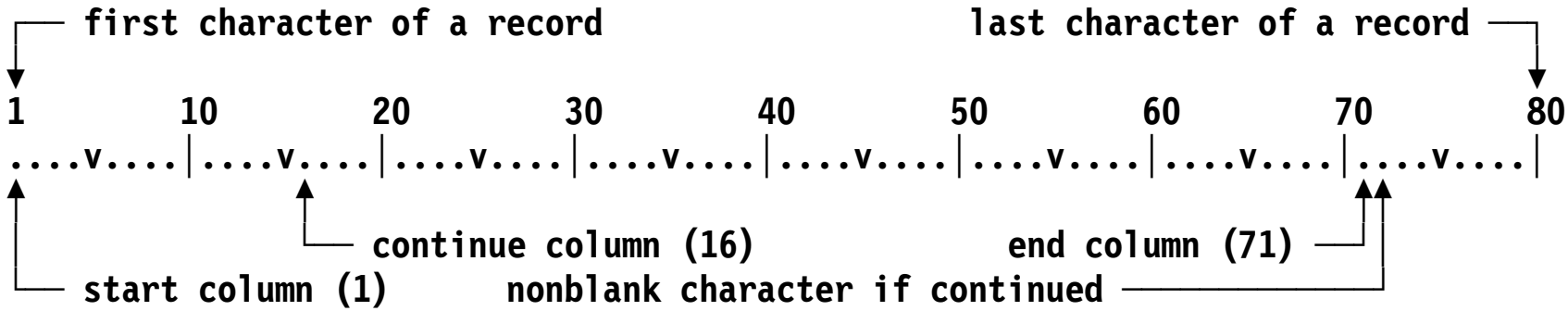
- The Assembler helps you prepare instructions for execution on System z
- Gives you maximum control over selection and sequencing of specific instructions
- Assembler Language itself is much simpler than other programming languages
- Main difficulties are
  - Learning an appropriate set of machine instructions for your applications
  - Learning all the auxiliary tools and programs needed to build and use Assembler Language programs

- Generally done in three stages:

  1. **Assembly;** The **Assembler** translates the statements of your *source program* into machine language instructions and data (“object code”) in the form of an *object module* for eventual execution by the CPU.
  2. **Linking:** The **Linker** combines your object module with any others required for satisfactory execution. The resulting *load module* is saved.
  3. **Program Loading:** The **Program Loader** reads your load module into memory and then gives CPU control to your instructions starting at the *entry point*.

  Your program then executes your instructions: reading, writing, and generating data

- Assembler Language statements are prepared on 80-byte “card image” records:



- Four types of statement:
  1. Comment: no object code generated; for clarification only. (Must have a ***** in the start column)
  2. Machine instruction: Assembler will generate object code for a CPU instruction
  3. Assembler instruction: a directive to the Assembler; may or may not cause generation of object code
  4. Macro instructions: you combine any of the four statement types into a group that can be invoked by name to generate other statements

- Non-comment statements have four fields (in left-to-right order):
  1. Name field: starts in column 1 (leftmost byte of source record), ends with first blank; usually optional
  2. Operation field: starts at least 1 blank after end of name field entry. Always required.
  3. Operand field: starts at least 1 blank after end of operation field entry. Usually required.
  4. Remarks field: starts at least 1 blank after end of operand field entry. Always optional.
- Typical practice: to improve readability, start each field in a fixed column (e.g. 1,10,18,40)

- First statement should be START, with your program name in the name field, 0 for the operand

```
MyProg1  START  0
```

- Following that should be some explanatory comment statements
- Last statement should be END, with the name of your program in the operand field

```
         END    MyProg1
```

  - The END statement only tells the Assembler to stop reading records; it doesn't tell your program to stop executing!

- An example of a small program with “Job Control” statements

```
                                  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
```

- “Job Control” statements (lines 1-3, 12) vary from system to system

- Six macro instructions are used extensively throughout the text:

PRINTOUT
Display contents of registers and named areas of memory.

READCARD
Read an 80-character record into memory.

PRINTLIN
Send a string of characters to a printer.

DUMPOUT
Display contents of memory in hexadecimal and character formats.

CONVERTI
Convert characters to a 32- or 64-bit binary integer.

CONVERTO
Convert a 32- or 64-bit binary integer to characters.

- Equivalent facilities may be available at your location.

Important elements of the Assembler Language; each has a numeric value

- Self-defining term: a constant value, writable in four forms (slide 10)
- Symbol: 1-63 characters
  - Value assigned by you *or* by the Assembler
  - Many uses in the Assembler Language

A constant value held by the Assembler in a two's complement 32-bit word

- **Decimal**: an unsigned string of decimal digits with value between 0 and $2^{31}-1$ (2147483647)
- **Hexadecimal**: a string of hexadecimal digits enclosed in X'...' with value between X'0' and X'FFFFFFFF' ($2^{32}-1$, 4294967295)
- **Binary**: a string of binary digits enclosed in B'...' with value between B'0' and B'11111111111111111111111111111111' ($2^{32}-1$, 4294967295)
- **Character**: a string of 1-4 characters enclosed in C'...', except that apostrophes (') and ampersands (&) are paired for each occurrence, as in C'''' and C'&&'
  - Term's value determined from EBCDIC representation of characters

- Every character is represented by a number

Char	Hex	Char	Hex	Char	Hex	Char	Hex
Blank	40	.	4B	a	81	b	82
c	83	d	84	e	85	f	86
g	87	h	88	i	89	j	91
k	92	l	93	m	94	n	95
o	96	p	97	q	98	r	99
s	A2	t	A3	u	A4	v	A5
w	A6	x	A7	y	A8	z	A9
A	C1	B	C2	C	C3	D	C4
E	C5	F	C6	G	C7	H	C8
I	C9	J	D1	K	D2	L	D3
M	D4	N	D5	O	D6	P	D7
Q	D8	R	D9	S	E2	T	E3
U	E4	V	E5	W	E6	X	E7
Y	E8	Z	E9	0	F0	1	F1
2	F2	3	F3	4	F4	5	F5
6	F6	7	F7	8	F8	9	F9

Ordinary symbols: 1-63 characters; first must be alphabetic

- Upper and lower case letters are equivalent: no distinction
- $, @, #, _ are treated as alphabetic
  - Safest to avoid using the first three in symbols
- Two types: external and internal (the most frequent form)
- Internal symbols have three key attributes (maintained by the Assembler):
  - Value
  - Relocatability
  - Length (not the number of characters in the symbol!)

- At assembly time, the Assembler doesn't know
  1. Where the Program Loader will put your program in memory
     - Managed by creating a model of the program, putting *relocation* data in the object module (Section 38)
  2. What other programs will the Linker combine with yours
     - Managed by using external linkages (Chapter X)
- **Relocation**: You (or the Assembler) assume the program starts at an *origin* (usually, 0)
  - Assembler calculates positions ("locations") of all assembled instructions and data relative to that origin
  - Program Loader *relocates* your locations, to addresses relative to the load address

- The Assembler uses a *Location Counter* (LC) to keep track of assembly-time positions in your program
  - Current LC value is represented by an asterisk (*****)
  - As instructions and data are generated, the Assembler adds the length of the generated data to form the location of the next item
    - Locations of instructions always rounded to an even location
- Important distinction:
  1. *Locations* refer to positions in the Assembler's model of your program
  2. *Addresses* refer to positions in memory at execution time

- Values are assigned to symbols in two ways:
  1. Name-field symbols usually take the current value of the Location Counter (before adding the length of generated data)

```
MyProg1  Start 0              Set assumed origin location 0
Start    BASR  15,0           Value of symbol "Start" is 0
```

  2. Sometimes symbol values are assigned by the programmer using an EQU statement

```
symbol   EQU   self-defining term      The most common form
ABS425   EQU   425                     ABS425 has value 425
Char_A   EQU   C'A'                    Char_A has value X'C1'
```

  3. The length of the generated data is usually assigned as the symbol's *length attribute*

- Symbols in high-level languages (usually called “variables”) have execution-time values

```
X <- 22./7. ;   /* Set X to an approximation to pi */
```

- Symbols in Assembler Language are used only at assembly time; they have **no** execution-time value (are NOT “variables”)
  - Used as names of places in a program that may contain execution-time values
  - Symbol values simply help to organize the program

- Typical machine instruction statement format:

```
symbol       operation   operand1,operand2,...        remarks
(optional)   (required)  ◄—— 0 to many ——►          (optional)
```

- Assembler Language operands are formed from *expressions*
  - Expressions are formed from *terms* and *operators*
- Operators are +, −, *, /
- Terms take several forms... (slide 18)

- A basic expression element is a *term*
  - A self-defining term (always absolute)
  - A symbol (absolute or relocatable)
  - A Location Counter reference ***** (always relocatable)
  - A Literal (always relocatable)
  - A symbol attribute reference (always absolute)
    - Length (`L'symbol`)
    - Integer (`I'symbol`)
    - Scale (`S'symbol`)

- An expression is an arithmetic combination of terms and operators
  7+4 X–C′X′ N/L′Item Size*Count
- A parenthesized expression is treated as a term
  (A+2)*(X′4780′**–JJ) (7)+(6–2)**
  - Parenthesized sub-expressions are evaluated first
- Unary (prefix) + and - are allowed

1. Each term is evaluated to 32 bits, and its relocatability is noted
2. Inner parenthesized sub-expressions evaluated first (from inside to out)
3. At the same level, do multiplication and division before addition and subtraction

   So, 2+5*3–6 —> (2+(5*3)–6), not ((2+5)*3)–6

4. No relocatable terms allowed in multiplication or division
5. For same-priority operations, evaluate from left to right

   So, 5*2/4 —> (5*2)/4, 5/2*4 —> (5/2)*4

6. Multiplication retains low-order 32 bits of 64-bit product

7. Division discards any remainder
   - Division by zero is allowed; result is zero (!)
8. Evaluation result is a 32-bit two's complement value
9. Relocatability attribute of expression determined from relocatability of terms:
   a. Pairs of terms with same attribute and opposite signs have no effect (they "cancel"); if all are paired, the expression is absolute
   b. One remaining unpaired term sets the attribute of the expression; + means simply relocatable, − means complexly relocatable
   c. More than one unpaired term means the expression is complexly relocatable (a rare occurrence)

- Machine-instruction statement operands have only one of these three forms (where "expr" = expression)

expr	**$expr_1$(expr$_2$)**	**$expr_1$(expr$_2$,expr$_3$)**
7	**8(7)**	**22(22,22)**
8*N+4	**A(B)**	**(A)((B),(C))**
(91)	**(91)(15)**	**(91)(,15)**

  - In the second and third forms, adjacent parentheses do *not* imply multiplication!
  - In the third form, $expr_2$ can be omitted if it is zero:

    $expr_1$(,expr$_3$) **[The comma is still required!]**

- Machine instructions: how you, the programmer, write them in Assembler Language
  - Mnemonics are brief descriptions of an instruction's action
- Examples of five basic instruction formats
  - Available operand types for each instruction format
    - $expr_1$ – absolute or relocatable
    - $expr_1(expr_2)$ – $expr_1$ absolute or relocatable, $expr_2$ absolute
    - $expr_1(expr_2,expr_3)$ – all three expr's absolute
  - Explicit and implied addresses
  - Explicit and implied lengths

- These are some commonly used RR-type instructions:

Op	Mnem	Instruction	Op	Mnem	Instruction
05	BALR	Branch And Link	06	BCTR	Branch On Count
07	BCR	Branch On Condition	0D	BASR	Branch And Save
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

- Typical operand field described as R_1,R_2 – operands of "expr$_1$" form

- Assembler must generate machine language form of the instruction:

- R_1 and R_2 designate first and second operand registers, *not* general registers 1 and 2!
- Since LR opcode is X'18':

```
LR    7,3          assembles to X'1873'
```

- Operands can be written as any expression with value 0 ≤ value ≤ 15

```
LR    3*4-5,1+1+1  also assembles to X'1873'
```

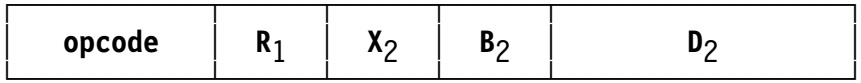
  - Assembly-time operands are *expressions* with value "7" and "3"
  - Execution-time operands are *contents* of general registers GR7 and GR3

- These are some commonly used RX-type instructions:

Op	Mnem	Instruction	Op	Mnem	Instruction
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
4D	BAS	Branch And Save	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

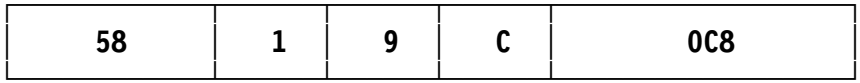
- RX-instruction first operand field described as R_1 (“$expr_1$” form)
- Second operand field described as S_2 (“$expr_1$” form), as $S_2(X_2)$ (“$expr_1(expr_2)$” form), as $D_2(X_2,B_2)$, (“$expr_1(expr_2,expr_3)$” form), or as $D_2(,B_2)$ (“$expr_1(,expr_3)$” form)

- Assembler must generate machine language form of the instruction:

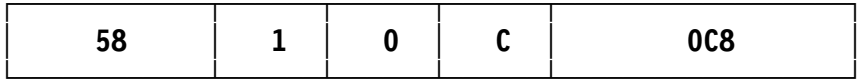


- First operand designates a general register
- Second operand usually designates a memory reference
  - B_2, D_2, and X_2 components used at execution time to calculate memory address (as described in Section 5)
  - Generic RX-instruction operands: R_1,*address-specification*
- Since L opcode is X'58',

```
L      1,200(9,12)        will generate
```




```
L      1,200(,12)         will generate
```



Explicit and Implied Addresses

- Two ways to create an *address-specification* operand:
  1. Explicit: you specify the base register and displacement
     - You provide the values in $D_2(X_2,B_2)$ or $D_2(,B_2)$
  2. Implicit: The Assembler assigns the base register and displacement for you
     - You specify an operand of the form S_2 or $S_2(X_2)$; the Assembler does assembly-time address resolution (described in Section 10)

- Examples of explicit addresses:

```
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
```

- General forms of RX-instruction second operands:

	Explicit Address	**Implied Address**
Not Indexed	$D_2(,B_2)$	S_2
Indexed	$D_2(X_2,B_2)$	$S_2(X_2)$

Typical RS- and SI-Type Instructions

- These are some typical RS- and SI-type instructions:

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
95	CLI	SI	Compare Logical Immediate	96	OI	SI	OR Immediate
97	XI	SI	Exclusive OR Immediate	98	LM	RS	Load Multiple
88	SRL	RS	Shift Right Single Logical	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

- *Many* ways to write their operand fields!

- RS-type instructions have two operand forms:
  - "RS-1" form, one register: $R_1,D_2(B_2)$ or R_1,S_2
  - "RS-2" form, two registers: $R_1,R_3,D_2(B_2)$ or R_1,R_3,S_2
- Assembler must generate machine language form of the instruction:

- R_1 operand designates a general register; R_3 operand can sometimes be omitted; $D_2(B_2)$ operand can be a memory reference or a number
- Examples of RS-type instructions:

```
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)
```


- Assembler must generate machine language form of the instruction:

- $D_1(B_1)$ (first) operand designates an *address-specification*
- Second (I_2) operand is an *immediate* operand
- General forms of SI-instruction operands:

	Explicit Address	Implied Address
SI	$D_1(B_1),I_2$	S_1,I_2

- Examples of SI-type instructions

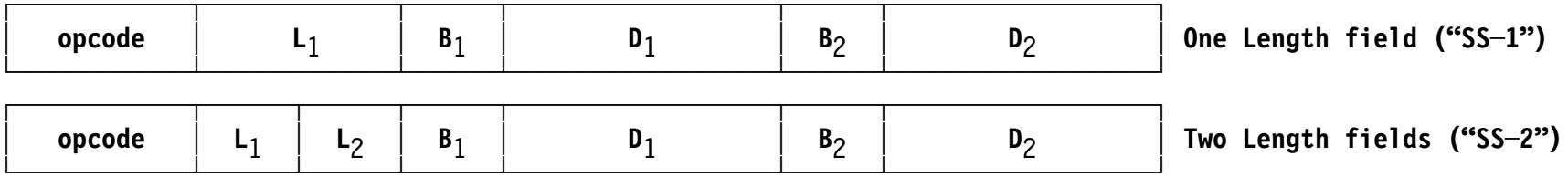
```
MVI   0(6),C'*'          Explicit D1(B1) address
CLI   Buffer,C'0'        Implied S1 address
```


- SS-type instructions have two memory operands, and one or two length operands
- These are popular SS-type instructions; the “Len” column shows the number of length fields in the instruction

Op	Mnem	Len	Instruction	Op	Mnem	Len	Instruction
D2	MVC	1	Move	D4	NC	1	AND
D5	CLC	1	Compare Logical	D6	OC	1	OR
D7	XC	1	Exclusive OR	DC	TR	1	Translate
F0	SRP	2	Shift And Round	F1	MVO	2	Move With Offset
F2	PACK	2	Pack	F3	UNPK	2	Unpack
F8	ZAP	2	Zero And Add	F9	CP	2	Compare
FA	AP	2	Add	FB	SP	2	Subtract
FC	MP	2	Multiply	FD	DP	2	Divide

- Instructions with F- opcodes operate on *packed decimal* data, discussed in Chapter VIII

- The Assembler generates two forms of SS-type machine instruction:



- Addresses *and* lengths can both be specified explicitly or implicitly.

SS-1 Form	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

- *You* write the length as N_1; the *Assembler* subtracts one to form L_1
- Some examples of SS-1 form instructions:

```
MVC   0(80,4),40(9)       Explicit length and addresses
CLC   Name(24),RecName    Explicit length, implied addresses
TR    OutCh(,15),7(12)    Implied length, explicit addresses
XC    Count,Count         Implied length and addresses
```


- For SS-2 form instructions, either or both operands may have explicit or implied addresses or lengths
  - This table shows some of the possible operand combinations:

SS-2 Form	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

- Some examples of SS-2 form instructions:

```
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
```


- Section 5 showed how the CPU creates Effective Addresses from addressing halfwords
- Now, we will see how the Assembler creates those addressing halfwords
- You supply the necessary information in a USING assembler instruction statement:

```
USING location,register
```

- **USING** is your promise to the Assembler:
  - ***If*** it assumes that *this* **location** is in *that* **register**, and calculates displacements and assigns base registers to addressing halfwords, ***then*** correct Effective Addresses will be generated at execution time.
- Understanding USING is important!

- A common method for establishing execution-time addressability uses the RR-type BASR ("Branch and Save") instruction

  BASR R_1,R_2

  - BASR puts the Instruction Address (IA) in the PSW into the R_1 register
  - This is the address of the *following* instruction
    - Remember: the IA was updated with the length of the BASR (2 bytes) during the fetch portion of the instruction cycle
  - If the R_2 operand is zero, nothing more is done
- The address in R_1 can be used as a *base address*
- R_1 can be used as a *base register*

- Suppose we assemble this little program fragment, and that we know it will be loaded into memory at address X'5000'

```
5000              START   X'5000'     Starting location
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'        Define Constant word integer 8
5028       ONE    DC      F'1'        Define Constant word integer 1
```

  - The length of each statement is added to its starting location (on the left)

- At *execution* time, after the BASR is executed, c(R6)=X'00005002'

  - Since the L instruction wants to refer to the word at X'5024', its displacement from X'5002 is X'5024'-X'5002'=X'022'

  - So if we create an addressing halfword X'6022' for the L instruction, we know that when the L is executed it will refer to the correct address

- We can continue this way...

- Suppose this fragment is to be loaded into memory at address X'84E8'

```
84E8              BASR     6,0          Establish base address
84EA       BEGIN  L        2,N          Load contents of N into GR2
84EE              A        2,ONE        Add contents of ONE
84F2              ST       2,N          STore contents of GR2 into N
          —twenty-two (X'16') additional bytes of instructions, data, etc.—
850C       N      DC       F'8'         Word integer 8
8510       ONE    DC       F'1'         Word integer 1
```

- At *execution* time, after the BASR is executed, c(R6)=X'000084EA'
  - Since the L instruction wants to refer to the word at X'850C', its displacement from X'84EA is X'850C' -X'84EA'=X'022'
  - So if we create an addressing halfword X'6022' for the L instruction, we know that when the L is executed it will still refer to the correct address. Completing all addressing halfwords yields this object code:

Address	Assembled Contents
84E8	0D60
84EA	58206022
84EE	5A206026
84F2	50206022
850C	00000008
8510	00000001

- We now know that it doesn't matter where the program is loaded, so we can assign base and displacement explicitly:

```
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'
```

- Computing displacements can be hard work! So we help the Assembler by using the values of symbols:

```
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'
```

- The Assembler calculates displacements for us; we assigned the base register

- So we tell the Assembler two items: the symbol **BEGIN** and register 6:

```
         USING BEGIN,6       Assume R6 will hold address of BEGIN
```

- The Assembler uses this to assign displacements and bases
- We now can write the program with *implied* addresses:

```
         BASR  6,0           GR6 will hold execution address of BEGIN
         USING BEGIN,6       Tell Assembler C(GR6)=address of BEGIN
BEGIN    L     2,N           Load c(N) into GR2
         A     2,ONE         Add c(ONE) to GR2
         ST    2,N           STore sum at N
*   ——— 22 bytes ———
N        DC    F'8'
ONE      DC    F'1'
```

  - And the Assembler does the hard work!

- The Assembler builds an accurate model of your program using the *Location Counter* (LC)
  - The position of each piece of object code (or reserved space) is given a *location* during assembly
  - The relative positions of all items in each major segment of a program is fixed at assembly time
- "*****" as a term has the value of the Location Counter, so we can use a *Location Counter Reference*

```
BASR  6,0          Establish base register
USING *,6          Tell Assembler base location is "here"
```

  - No need to define a symbol just for use in the USING statement; specifying a symbol on the instruction following a BASR is (a) inconvenient, (b) unnecessary, and (c) sometimes a poor idea

- Suppose we make a mistake in the L instruction named **BEGIN**:

```
Location   Object Code     Statement
  0000     0D60                BASR   6,0
                               USING  BEGIN,6
  0002     58606022    BEGIN   L      6,N     ←Wrong register! (6, not 2)
  0006     5A206026            A      2,ONE
  000A     50206022            ST     2,N
                          ------------------------
  0024     00000008    N       DC     F'8'
  0028     00000001    ONE     DC     F'1'
```

  - The program *assembles* correctly, but won't *execute* correctly!
    - Suppose it is loaded into memory starting at address X'5000'
  - When the L instruction is fetched, c(GR6) = X'00005002'
  - When the L instruction is executed, c(GR6) = X'00000008'
  - When the A instruction is executed, its Effective Address is X'0000002E' (not X'00005028'!)
  - Worse: the ST will try to store into memory at address X'0000002A', probably causing a memory protection exception
- Be *very* careful not to alter the contents of base registers!

- The Assembler scans the program twice; the first time (“Pass 1”):
  - Each statement is read
  - Lengths of instructions, data, and reserved areas are determined, locations are assigned
  - Symbols whose positions are known are given location values
    - Some symbols may appear as operands before having a value
  - No object code is generated in Pass 1
- At END statement, all symbols and values should be known (“defined”)
  - If not, various diagnostic messages are created

- Values of expressions can be calculated from known symbol values
- USING statement data is entered in the USING Table. For example:

basereg	base_location	RA
6	00000002	01

- For instructions with implied addresses:

  displacement = (implied_address value) – (**base_location value)**

- Relocatability Attribute (RA) of an implied address expression must match RA of a USING Table entry

- If successful, the Assembler has *resolved* the implied address.
  If not, the implied address is *not addressable*
- The Assembler does at assembly time the reverse of what the CPU does at execution time:

  Assembly: displacement = (operand_location) – (base_location)
  Execution: (operand address) = displacement + (base address)

- Suppose there is more than one USING statement:

```
Location  Statement
  0000            BASR      6,0
                  USING     *,6         Original USING statement
  0002    BEGIN   L         2,N
                  USING     *,7         Added USING statement
  0006            A         2,ONE
          ——————— as before ————————
```

- The USING Table now looks like this:

basereg	base_location	RA
6	00000002	01
7	00000006	01

- When the A statement is assembled, two addressing halfwords are possible:
  - With register 6: X'00000028'-X'00000002' = X'026' with addressing halfword X'6026'
  - With register 7: X'00000028'-X'00000006' = X'022' with addressing halfword X'7022'
- The Assembler chooses the resolution with the *smallest* displacement

- To remove USING Table entries, use the **DROP** statement

```
DROP  register
```

  - In the previous examples, if we write

```
DROP  6
```

  - then the USING Table looks like this:

basereg	base_location	RA
	empty	
7	00000006	01

- If a DROP statement is written with *no* operand, all USING Table entries are removed:

basereg	base_location	RA
	empty	
	empty	

- Many conditions can cause addressability problems

1. For instructions with 12-bit unsigned displacement fields, the value of a displacement must lie between 0 (X'000') and 4095 (X'FFF')
   - References requiring displacements outside that range are *not addressable*

```
SetBase  BASR  6,0
         USING *,6
         L     0,*+5000      Would require positive displacement X'1388'
         L     0,SetBase     Would require negative displacement X'FFFFFFFE'
```

2. If the USING Table is empty, implied addresses can't be resolved
   - Except for resolutions of absolute implied addresses with register 0; see slide 48
3. Symbols in other sections have different Relocatability Attributes
   - Techniques used to refer to them are described in Chapter X
4. Complexly relocatable operands (rare!) are never addressable

- The Assembler has an implied USING Table entry for register 0:

basereg	base_location	RA
0	00000000	00
	etc.	

(Absolute expressions have RA = 00)

- If an operand is (a) absolute and (b) between 0 and 4095 in value, the Assembler can resolve it to an addressing halfword with base register zero

```
LA    7,100          100 = X'064', so addressing halfword = X'0 064'
LA    7,4000         4000 = X'FA0', so addressing halfword = X'0 FA0'
```

- You can specify an absolute base_location in a USING statement:

```
LA    9,400
USING 400,9          Base address = 400 = X'190'
LA    3,1000         1000 = X'3E8'
```

  - Two resolutions are possible: addressing halfwords X'03E8' and X'9258'
    - The Assembler chooses the one with the smaller displacement: X'9258'

- The Assembler uses these rules for resolving USING-based addressing:

  1. The Assembler searches the USING Table for entries with a Relocation Attribute matching that of the implied address.
     - (It will almost always be simply relocatable, but may be absolute.)
  2. For all matching entries, the Assembler tries to derive a valid displacement. If so, it selects as a base register the register with the smallest displacement.
  3. If more than one register yields the same smallest displacement, the Assembler selects the highest-numbered register.
  4. If no resolution has been completed, and the implied address is absolute, the Assembler tries a resolution with register zero and base zero.

- Section 11 describes the Assembler's basic data definition instruction, DC ("Define Constant").
- 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 position 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.

- Section 11 describes DC-statement rules for defining constants of any type
  - Fixed-point binary data: signed and unsigned; 16-, 32-, and 64-bit lengths
  - Logical data, binary and hexadecimal: 1 to 256 byte lengths
  - Address-valued data: 3, 4, and 8 byte lengths
  - Character data: 1 to 256 bytes, in EBCDIC, ASCII, Unicode, and Double-Byte formats
  - Decimal data: 1 to 16 bytes, in zoned and packed decimal formats
  - Floating-point data: 4, 8, or 16 bytes, in hexadecimal, binary and decimal formats
- **Note:** DC defines data with *initial* values, not *unchangeable* “constants”.
  - A program can change those values!

- The constant **DC F'8'** generates a 4-byte binary integer X'00000008' on a word boundary
- A DC statement specifies at least 4 items: [[comments for F'8']]
  1. The *type of conversion* from external to internal representations [[decimal to binary]]
  2. The *nominal value* (external representation) of the constant [[decimal 8]]
  3. The *length* of the constant [[4 bytes]]
  4. The *alignment* of the constant [[word]]
- Examples of four constant types:

```
DC    F'8'          Word binary integer
DC    C'/'          Character constant
DC    X'61'         Hexadecimal constant
DC    B'01100001'   Binary constant
```

  - Note that the last three constants use the same nominal value representation as the corresponding self-defining terms.

- DC statements can use all statement fields; “DC” and “operand(s)” are required

```
<name>   DC   <operand(s)>   <remarks>
```

- Each operand may have up to 4 parts, in this order:
  1. Duplication factor (optional; defaults to 1)
  2. Type (1 or 2 letters; required)
  3. Zero to several modifiers (optional)
  4. Nominal value in external representation, enclosed in delimiters (required)
     - Delimiters are apostrophes or parentheses, depending on type

- Many constants have natural boundary alignments (such as type F)
  - The Assembler will round up the LC (if needed) to place a constant on the proper boundary
  - Bytes skipped for alignment are normally filled with X′00′ bytes
- Automatic alignment is not performed if
  1. It isn′t needed: that is, the LC happens to fall on the desired boundary
  2. The type of constant specified doesn′t require alignment, such as types C, B, or X (among others)
  3. A length modifier is present, which suppresses alignment

- A length modifier specifies a constant's exact length (within limits)
- Written as the letter **L** followed by a nonzero decimal integer or parenthesized positive absolute expression:

```
Ln        or        L(expr)
```

- Examples:

```
A       DC    FL3'8'          Generates X'000008' at current location
B       DC    FL(2*4-5)'8'    Generates X'000008' at current location
C       DC    F'8'            Generates X'00000008' at next word boundary
D       DC    FL4'8'          Generates X'00000008' at current location
```

- Symbols **A, B, D** are given values of the LC where the constants are generated; symbol **C** is given the LC value *after* bytes are skipped for alignment

- You can generate copies of a constant in several ways:
  - Multiple operands

```
DC    F'8',F'8',F'8'
```

  - Multiple statements:

```
DC    F'8'
DC    F'8'
DC    F'8'
```

  - Duplication factors:

```
DC    3F'8'
```

- Duplication factors can be used on each operand:

```
DC    2F'8',2F'29',F'71',3F'2'      8 word constants
```


- Almost all constant types accept multiple nominal values, separated by commas

```
DC    F'1,2,3,4,5'        Five word constants
DC    X'A,B,C,D,7'        Five one-byte constants
DC    B'001,010,011'      Three one-byte constants
```

- Many constant types accept embedded spaces for readability:

```
DC    F'1 000 000 000'    Easier than counting adjacent zeros
DC    X'1 234 567 89A'    Five-byte constant
```

- Character constants are the exception: a comma or a space is part of the nominal value

```
DC    C'1,2,3,4,5'        One nine-byte constant
DC    C'1 2 3 4 5'        One nine-byte constant
```


- Every symbol has a Length Attribute (LA)
  - Assigned by you, or by the Assembler (most usually)
- LA of symbols naming instructions is the length of the instruction

```
LOAD     LR    7,3              LA of LOAD  = 2
BEGIN    L     2,N              LA of BEGIN = 4
```

- LA of symbols naming DC statements is the length of the *first* generated constant, *ignoring* duplication factors

```
Implied  DC    F'8'             LA of Implied  = 4
Explicit DC    XL7'ABC'         LA of Explicit = 7
Multiple DC    3F'8'            LA of Multiple = 4
List     DC    F'1,2,3'         LA of List     = 4
OddOnes  DC    B'1',F'2',X'345' LA of OddOnes  = 1
```

- For almost all EQU statements, the Assembler assigns LA 1:

```
R7       Equ   7                LA of R7       = 1
```


Decimal Exponents

- Very large numbers may have many trailing zeros
  - A Decimal Exponent can simplify writing such constants
  - Write “En” after the leading (nonzero) digits of the nominal value, where “n” is the desired number of added zeros

```
BillionA DC    F'1000000000'        The hard way
BillionB DC    F'1 000 000 000'     An easier way
BillionC DC    F'1E9'               The easiest way
```

- You can even write constants with negative exponent values “n”

```
HundredA DC    F'1E2'               Generates X'00000064'
HundredB DC    F'1000E-1'           Generates X'00000064'
```

- Exponent *modifiers* apply to all nominal values in the operand

```
Hundreds DC    FE2'1,2,3,4'         Constants for 100, 200, 300, 400
Hundreds DC    FE1'1E1,2E2,3,4E1'   Constants for 100, 2000, 30, 400
```

- Decimal exponents and exponent modifiers are often used in floating-point constants

- Section 12 provides more detail about seven basic constant types:
  - **F** Two′s complement binary integers, normally word length
  - **H** Two′s complement binary integers, normally halfword length
  - **A** Address- or expression-valued, normally word length
  - **Y** Expression-valued, normally halfword length
  - **C** Character-valued constant, length 1 to 256 bytes
  - **B** Bit-valued constant, length 1 to 256 bytes
  - **X** Hexadecimal-valued constant, length 1 to 256 bytes
- Literals provide a powerful way to define *and* address constants

- F-type and H-type constants generate binary values, with default word and halfword lengths respectively.

```
DC    F'-10'          Generates X'FFFFFFF6', word aligned
DC    H'-10'          Generates X'FFF6', halfword aligned
```

- If a length modifier is present, the two types are identical:

```
DC    FL5'-10'        Generates X'FFFFFFFF6', unaligned
DC    HL5'-10'        Generates X'FFFFFFFF6', unaligned
```

- To extend the range of F- and H-type constants by one bit, you can generate *unsigned* constants
  - Write the letter **U** before the nominal value

```
DC    F'U2147483648'  Generates X'80000000', word aligned (2**31)
DC    H'U65535'       Generates X'FFFF', halfword aligned (2**16-1)
DC    F'U4294967295'  Generates X'FFFFFFFF', word aligned (2**32-1)
DC    H'1,U2'         Generates X'00010002', Mixed forms
```


- The address constant (or “adcon”) is useful in many contexts.
  - A-type defaults to word length (explicit lengths 1, 2, 3, 4 bytes);
    Y-type defaults to halfword length (explicit lengths 1, 2 bytes)
  - The nominal value can be an absolute or relocatable expression:

```
R7      Equ   7              Define a symbol value
Expr1   DC    A(C'A'+48)     Generates X'000000F1', word aligned
        DC    A(R7)          Generates X'00000007', word aligned
        DC    A(Expr1)       Will contain execution-time address of Expr1
        DC    AL1(*-Expr1)   Will generate X'0C', unaligned
Here    DC    A(*+64)        Will contain execution-time address of Here+64
```

- Y-type constants are rarely used now, and only for absolute expressions

```
Expr1   DC    Y(C'A'+48)     Generates X'00F1', halfword aligned
        DC    Y(R7)          Generates X'0007', halfword aligned
```

  - Early (and very small) machines used relocatable 16-bit address constants
- The ability to generate constants from expressions is very powerful

- C-, B-, and X-type constants can be up to 256 bytes long
  - If you don't specify an explicit length, the Length Attribute (LA) of symbols naming such constants is the constant's *implied* length: the number of bytes generated for the first operand (ignoring duplication factors)

```
A       DC    C'12345'       Generates X'C1C2C3C4C4'; LA of A = 5
B       DC    X'123456'      Generates X'123456';     LA of B = 3
C       DC    2B'10100101'   Generates X'A5A5';       LA of C = 1
```

- Apostrophes and ampersands in C-type constants must be *paired* for each single occurrence in the generated constant

```
        DC    C''''          Generates X'7D'
        DC    C'&&&&'        Generates X'5050'
```


- The space allocated for a constant is defined either by default or by a length modifier
- If the constant is too small for the space, it must be *padded*; if the constant is too large for the space, it must be *truncated*
- The Assembler's actions in such cases depends on the constant type

Type	Too Small	Too Large
F,H	Pad with sign bits on left	Truncate on left; error message
A,Y	Pad with sign bits on left	Truncate on left; error message
C	Pad with spaces on right	Truncate on right
B	Pad with zero bits on left	Truncate on left
X	Pad with zero digits on left	Truncate on left

- A literal is a type of symbol that references *and* defines a constant
- Written as an equal sign followed by a DC operand

```
=F'8'      =H'22'      =4X'40'      =CL2'$'      =A(X'40',C'$')
```

- Some limitations and restrictions:
  - Multiple operands are not allowed (but multiple values are OK)
  - Duplication factors are allowed, but may not be zero
  - Literals are not allowed as operands of address constants
- The Assembler tries to diagnose instructions that can directly modify a literal

```
L     2,=F'8'       Valid reference
ST    2,=F'8'       Invalid; tries to modify the literal
```

- A literal is more likely to be a “constant” constant

- The Assembler collects literals internally as they are referenced; they must be assembled somewhere into your program
- The LTORG ("Literal Origin") instruction lets you specify where the collected literals should be placed
  - It's important that all literals are addressable!
- Literals are placed in the program in order of decreasing alignment requirement
  - Assembler's internal collection is emptied
- Subsequent literal references start a new collection
  - A subsequent LTORG will generate the new ones
    - The same literal can appear more than once; they are treated as distinct symbols
- At the END statement, all remaining literals are generated

- As System/360 evolved into System z, many Assembler enhancements have been needed
- Constants (numeric, address-valued, character) were extended
  D type extension defaults to doubleword length, alignment

```
DC      FD'1E15'        X'00038D7EA4C68000'
DC      AD(C'ABC')      X'0000000000C1C2C3'
```

- Character constants accommodate all three representations
  A type extension converts EBCDIC nominal value to ASCII
  U type extension converts EBCDIC nominal value to Unicode UTF-16
  E type extension generates the original EBCDIC nominal value, even if the Assembler's TRANSLATE option specifies an arbitrary conversion

```
DC      C'ABC'          X'C1C2C3'           EBCDIC (TRANSLATE-able)
DC      CE'ABC'         X'C1C2C3'           EBCDIC always
DC      CA'ABC'         X'414243'           ASCII  always
DC      CU'ABC'         X'004100420043'     Unicode UTF-16
```


- Section 13 shows ways to define and organize data and work areas
  - The DS (“Define Storage”) instruction is similar to DC, but generates no object code
  - The EQU (“Equate”) instruction lets you assign values to symbols, or define similarities of one symbol to another
  - The ORG (“Set Origin”) instruction lets you adjust the position of the Location Counter
- With combinations of these instructions, you can define data structures that greatly simplify many programming tasks

- DS is very similar to DC, except that (a) no object code is generated, (b) no nominal value is required in operands

```
DS    F          Both statements allocate 4 bytes of ...
DS    F'8'       uninitialized space on a word boundary
```

- Multiple operands and values are allowed

```
DS    F'1',X'ABC',C'ABC'  Allocates 9 bytes, word aligned
DS    H'2,4,6,8'          Allocates 8 bytes, halfword aligned
```

- As with DC, gaps can appear due to boundary alignment

```
DS    F,X          Allocate 5 bytes, word aligned
DS    F            Skip 3 bytes for word alignment
```

- Length Attributes of names are derived from first operand

```
Area1   DS    80C          Allocate 80 bytes; LA of Area1 = 1
Area2   DS    CL80         Allocate 80 bytes; LA of Area2 = 80
```

  - Both statements allocate 80 bytes, unaligned

- Zero duplication factor can be used with DS *and* DC
  - Boundary alignment and symbol attributes done “as usual”

```
DC    0F'8'              Word alignment, nothing generated
```

  - Only difference is treatment of alignment gaps
    - In DS statements, gaps are uninitialized
    - In DC statements, gaps are uninitialized if the byte preceding the gap was uninitialized; otherwise the gap is filled with X'00' bytes

```
DC    F'23',X'BE'        5 bytes on a word boundary
DC    0F'8',F'47'        3-byte gap filled with X'00'
```

- Useful for overlaying related fields. Example: U.S. telephone number

```
PhoneNum DS    0CL10      Full ten digits of the number
AreaCode DS    CL3        Three digits of area code
Prefix   DS    CL3        Locality prefix
LocalNum DS    CL4        Local number
```

  - You can refer to the full, or individual, fields as needed

- The basic form of EQU is

```
symbol   EQU   expression
```

- "symbol" receives the value, relocatability, and length of "expression"
  - If we write

```
A        DC    F'8'
B        EQU   A
```

  - Then B will have the same value, relocatability, and length attributes as A
- Assigning an absolute expression is very useful. For example:

```
NItems   EQU   75            Number of table items (Note: not F'75')
Count    DC    A(NItems)     Constant with number of table items
Before   DS    (NItems)F     Space for "NItems" words
After    DS    (NItems)F     (Not "75F")
```

  - If a change must be made to the size of the tables, *only* the EQU statement needs updating before re-assembly

- Extended EQU syntax supports up to 5 operands
  (the last two are used for conditional assembly and macros)

```
symbol   EQU   value,length,type[,program-attribute,assembler-attribute]
```

  - **value** operand: its value, relocatability, and length are assigned to **symbol**
  - **length** is assigned to **symbol**, overriding any previous length assigned from **value**
  - **type** is assigned to **symbol**. If no **type** operand is present, the Assembler assigns type U (“Unknown”)

- Extended syntax is usually used with just the first two operands
- We can rewrite the Phone Number example to use extended syntax:

```
PhoneNum DS    CL10                Space for entire number
AreaCode Equ   PhoneNum,3,C' C'    Overlay AreaCode
Prefix   Equ   AreaCode+3,3,C' C'  Overlay Prefix
Local_No Equ   Prefix+3,4,C' C'    Overlay Local_No
```


- The ORG instruction modifies the Location Counter by setting its value to its operand expression:

```
        ORG   relocatable_expression
```

- We can revise the Phone Number example to use ORG:

```
PhoneNum DS    CL10              Space for entire number
         ORG   PhoneNum          Reposition Location Counter
AreaCode DS    CL3               Define AreaCode
Prefix   DS    CL3               Define Prefix
Local_No DS    CL4               Define Local_No
```

- ORG also supports an extended syntax:

```
        ORG   relocatable_expression,boundary,offset

        ORG   *,8,+6   Set to 2 bytes before a doubleword boundary
```

  - The LC is first set to the **expression** value; then it is rounded up to the next power-of-two **boundary**; and finally the **offset** is added
    - Exercise: Why **+6**? Why not **ORG *,8,-2** ?

- Parameterization uses a small number of values to define and control constants, data areas, field lengths, offsets, etc.
  - Many examples of parameterization were shown on previous slides
- Suppose we must read, modify, and write 80-byte records

```
RecLen   Equ   80            Record length (for now)
InRec    DS    CL(RecLen)    Area for input records
WorkRec  DS    CL(Reclen)    Work area for records
OutRec   DS    CL(RecLen)    Area for output records
```

  - If the record length is changed, only the EQU statement needs updating
- Suppose a table of 45 records must be maintained in storage

```
NRecs    Equ   45              Number of records in storage
RecNum   DC    Y(NRecs)        Constant with number of records allowed
StorRecs DS    (NRecs-1)CL(RecLen)   Space for all but one records
LastRec  DS    CL(RecLen)      Last record in storage
```

  - Changing the number of records and the allocated space is a simple update
- Parameterization simplifies *many* aspects of programming!
  - It improves program readability and understandability

- Address constants usually refer to locations internal (or external) to a program
- They can also be used to generate tables of constants
  - Example: table of byte integers from 0 to 10:

```
IntTbl   DC    FL1'0,1,2,3,4,5,6,7,8,9,10'  Generates 0,1,...9,10
           or
IntTbl   DC    11AL1(*-IntTbl)              Generates 0,1,...9,10
```

- In constants with a duplication factor and * in the nominal value, the nominal value is *re-evaluated* as each constant is generated
  - Example: table of byte integers from 10 to 0:

```
IntTbl2  DC    FL1'10,9,8,7,6,5,4,3,2,1,0'  Generates 10,9,...1,0
           or
IntTbl2  DC    11AL1(IntTbl2-*+10)          Generates 10,9,...1,0
```

- Very complex tables can be created using such techniques
- Exercise: Which **IntTbl**s are easier to expand?

- Slide 24
  - Suppose the LC is already at a doubleword boundary: then the offset -2 might back up the LC over existing object code or data areas.
- Slide 26
  - The ones with AL1-type constants only need to modify the duplication factor, not the nominal value as in the FL1-type constants.
    - But for the example using **IntTbl2**, you'll need to change the +10 to one less than the duplication factor.
    - A better way:

```
NumInt2  EQU   11                    Number of generated values
IntTbl2  DC    (NumInt2)AL1(IntTbl2-*+NumInt2-1)
```

      - Now, only the EQU statement needs changing

Chapter V: Basic Instructions 1

This chapter introduces basic instructions used in many Assembler Language programs

- Section 14 discusses instructions that move data among the general registers (GRs), and between the registers and memory
- Section 15 describes the important “Branch on Condition” instructions that let your programs select alternate instruction paths
- Section 16 covers instructions for binary addition, subtraction, and comparison of signed and unsigned operands
- Section 17 examines instructions that shift binary operands in the GRs
- Section 18 reviews instructions that multiply and divide binary operands in the GRs
- Section 19 introduces instructions that perform the AND, OR, and XOR logical operations on bit groups in the GRs

This section describes instructions that move operands among general registers, and between general registers and memory

- Data operands can be 1, 2, 4, or 8 bytes long
  - For some instructions, operands can be 0-4 bytes long
- Register operands can be 32 or 64 bits wide
  - For some instructions, operands can be 1-4 bytes long
- Some instructions will sign-extend the high-order bit of a source operand to fit the length of the target register
- Some instructions can test the value of an operand, or complement its value

The Load (**L**) and Store (**ST**) instructions move data from memory to a GR (**L**) and from a GR to memory (**ST**)

- Both are RX-type instructions
  - The memory address is an indexed Effective Address
- Neither *requires* word alignment of the memory address
  - But it's advisable for many reasons (performance, access exceptions, ...)
- Only the right half of the GR (bits 32-63) is involved; bits 0-31 are ignored
- Examples:

```
L     7,-F'-97'      c(GR7) replaced by X'FFFFFF9F'
ST    7,Num          c(NUM) replaced by c(GR7), GR7 unchanged
```


The RS-type Load Multiple (**LM**) and Store Multiple (**STM**) instructions let you load and store a *range* of GRs in a single operation

- Rather than write

```
L   1,A                      ST   1,B
L   2,A+4        and         ST   2,B+4
L   3,A+8                    ST   3,B+8
```

you can write

```
LM  1,3,A        and         STM  1,3,B
```

- The instruction format is

```
LM (or STM)    R1,R3,D2(B2)      (explicit address)
LM (or STM)    R1,R3,S2          (implied address)
```

- The register contents are transmitted between GRs and successive words in memory, starting with the R_1 register and ending with the R_3 register
  - If R_3 is smaller than R_1 then GRs R_1-15 are transmitted followed by GRs 0-R_3
- These instructions are often used for “status preservation”

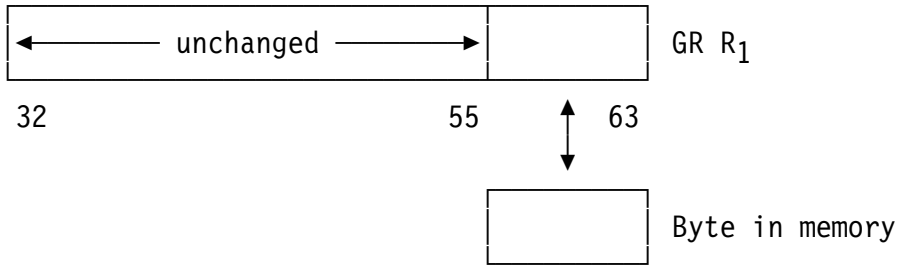
- The RX-type Load Halfword (**LH**) and Store Halfword (**STH**) instructions transfer two bytes between memory and the *rightmost* 2 bytes of a GR
  - STH simply stores the 2 bytes at the Effective Address
  - LH puts the 2 bytes in the right end of the GR, *and* sign-extends the leftmost through the rest of the GR

  - If the value is in the range $-2^{15} \leq$ value $< 2^{15}$, no data is lost; but:

```
L     0,=F'65537'    c(GR0)=X'00010001'  +65537 = 2**16+1
STH   0,A            c(A)  =    X'0001'  Lost a bit!
LH    1,A            c(GR1)=X'00000001'  Lost significance!

L     0,=F'65535'    c(GR0)=X'0000FFFF'  +65535 = 2**16–1
STH   0,A            c(A)  =    X'FFFF'  No lost bits, but wrong sign
LH    1,A            c(GR1)=X'FFFFFFFF'  (–1!) Lost significance!
```


- The RX-type Insert Character (**IC**) and Store Character (**STC**) instructions transfer a byte between memory and the *rightmost* byte of a GR
- For IC, the remaining bytes of the GR are unchanged



- Two examples:

```
(1)     L     0,=F'-1'      c(GR1) = X'FFFFFFFF'
        IC    0,=C'A'       c(GR1) = X'FFFFFFC1'

(2)     IC    0,X           Get 1st byte of c(X)
        STC   0,Y+1         Store at 2nd byte of Y
        IC    0,X+1         Get 2nd byte of c(X)
        STC   0,Y           Store byte at Y
        - - -
X       DC    C'AB'
Y       DS    CL2           c(Y) becomes C'BA'
```


- The RS-type Insert and Store Multiple Characters (**ICM** and **STCM**) instructions are generalizations of **IC** and **STC**
- You can specify any or all bytes of a 32-bit GR

```
ICM (or STCM)    R1,M3,D2(B2)    (explicit address)
ICM (or STCM)    R1,M3,S2        (implied address)
```

  - Bits positions of the M_3 "mask" operand specify which GR bytes participate
  - Bytes from memory are successive bytes at the Effective Address (none are skipped)
  - ICM sets the Condition Code:

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

- Example: suppose c(GR1) = X'AABBCCDD'

```
ICM   1,B'0101',=X'1122'      c(GR1) now = X'AA11CC22', CC=2
STCM  1,B'1010',Z             c(Z)   now = X'AACC'
```


- Many instructions copy data among registers; some complement, extend, or test the operand; the R_1 and R_2 operands need not differ

Mnemonic	Action	CC Values
LR	$c(GR\ R_1) \leftarrow c(GR\ R_2)$	Unchanged
LTR	$c(GR\ R_1) \leftarrow c(GR\ R_2)$	0,1,2
LCR	$c(GR\ R_1) \leftarrow -c(GR\ R_2)$	0,1,2,3
LPR	$c(GR\ R_1) \leftarrow \lvert c(GR\ R_2)\rvert$	0,2,3
LNR	$c(GR\ R_1) \leftarrow -\lvert c(GR\ R_2)\rvert$	0,1

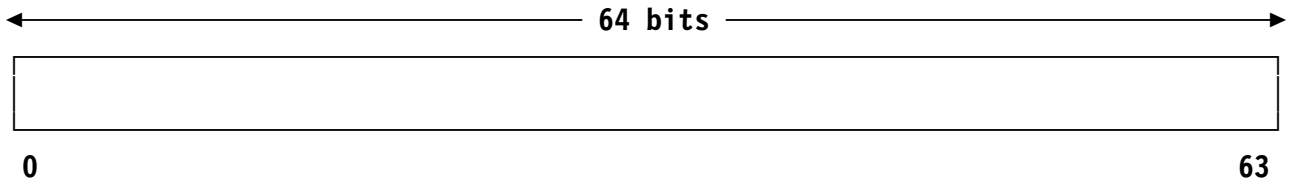
  - CC settings:

```
CC = 0        Result is zero
CC = 1        Result is negative, < 0
CC = 2        Result is positive, > 0
CC = 0        Result has overflowed
```

- Examples

```
LR    0,1          c(R0) ← c(R1), CC unchanged
LTR   1,1          Test c(R1), set CC
LCR   2,1          c(R2) = −c(R1), set CC
LNR   3,1          c(R2) = −|c(R1)|, set CC
```


- 64-bit GRs use many equivalent and extended 32-bit instructions



- Some instructions use all bits of a 64-bit GR:

```
LG    R1,S2        Load a doubleword into 64-bit GR
STG   R1,S2        Store a doubleword from 64-bit GR
LMG   R1,R3,S2     Load doublewords into 64-bit GRs
STMG  R1,R3,S2     Store doublewords from 64-bit GRs
```

- Some instructions use only the leftmost 32 bits of a 64-bit GR:

```
LMH   R1,R3,S2     Load words into left halves of 64-bit GRs
STMH  R1,R3,S2     Store words from left halves of 64-bit GRs
ICMH  R1,M3,S2     Insert characters into left half of GR
STCMH R1,M3,S2     Store characters from left half of GR
```

- Some of these instructions use RXY or RSY format:

opcode	R_1	X_2	B_2	DL_2	DH_2	opcode
opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

  - Their 20-bit base-displacement format is described in Section 20

- These instructions are similar to the 32-bit forms, but now have **G** in the mnemonic. (We sometimes use “GR” for 32-bit registers, and “GG” for 64-bit registers.) The CC settings are as shown on slide 8

Mnemonic	Action	CC Values
LGR	$c(GG\ R_1) \leftarrow c(GG\ R_2)$	Not changed
LTGR	$c(GG\ R_1) \leftarrow c(GG\ R_2)$	0,1,2
LCGR	$c(GG\ R_1) \leftarrow -c(GG\ R_2)$	0,1,2,3
LPGR	$c(GG\ R_1) \leftarrow \lvert c(GG\ R_2) \rvert$	0,2,3
LNGR	$c(GG\ R_1) \leftarrow -\lvert c(GG\ R_2) \rvert$	0,1

- Examples:

```
*       Assume c(GG2) = +1, c(GG3) = 0
        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
```


- These instructions can replace an equivalent pair:

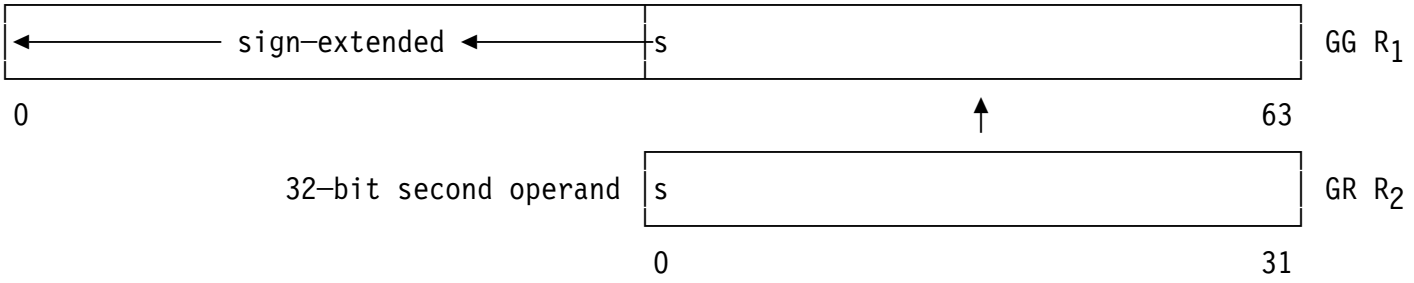
```
          LT    R1,S2
replaces
          L     R1,S2
          LTR   R1,R1
or even
          ICM   R1,B'1111',S2

and,

          LTG   R1,S2
replaces
          LG    R1,S2
          LTGR  R1,R1
```

- Condition Code settings are the same as for ICM
- Note that L and LG are indexable instructions, but ICM is not
- ICM and ICMH cannot perform the function of LTG, because they set the CC separately for each half of GG R_1

These instructions automatically sign-extend a 32-bit operand to 64 bits



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

- Example

```
LCGFR 0,1     is  equivalent to    LGFR  0,1
                                   LGCR  0,0
```

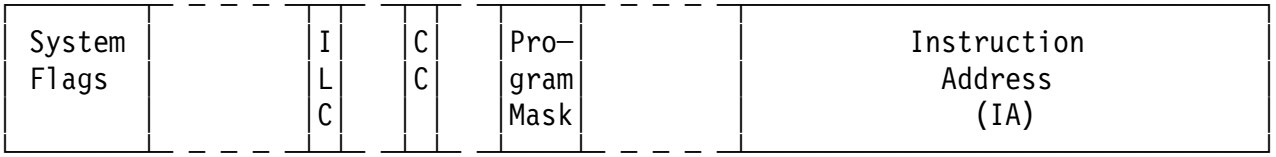

- These instructions simplify occasional programming tasks:
- Load Byte: insert a byte into the rightmost R_1 register byte and sign-extend its leftmost bit (**LB, LBR; LGB, LGBR**)
- Load Logical Character: insert a byte into the rightmost R_1 register byte and zero the remaining register bits (**LLC, LLCR; LLGC, LLGCR**)
- Load Logical Halfword: insert 2 bytes into the rightmost R_1 register 2 bytes and zero the remaining register bits (**LLH, LLHR; LLGH, LLGHR**)
- Load Logical (Word): load the right half of a 64-bit register and zero the left half (**LLGF, LLGFR**)
- Load Logical Thirty One Bits: load the right half of a 64-bit register; zero its leftmost bit, and also zero the left half (**LLGT, LLGTR**)
  - Reasons for this unusual behavior are discussed in Sections 20 and 37

- Some beginners make misleading assumptions about Assembler Language and System z instructions

1. Since both L (Load) and LR (Load Register) load a general register, they must be equivalent.
2. Since L (Load) and ST (Store) are complementary instructions, then for LR (Load Register) there must be a STR (Store Register) instruction.
   - There's no STR; just use LR with reversed operands
3. Because programmers often define symbols like `R0, R1, ... R15` to refer to the general registers, it's easy to assume that a symbol like `R1` always *means* GR1.
   - But `R1` is just a name for a number! You could just as well write

```
R92     EQU   1           "R92" means Register 92 ??
        L     R92,XYZ     You could write something like this...
        L     1,XYZ       and get the same result as writing this!
```


- Many instructions set the value of the PSW's 2-bit Condition Code (CC)

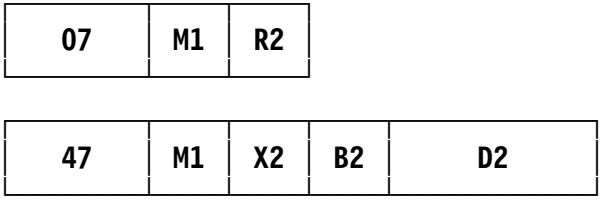


  - Possible CC values are 0, 1, 2, 3
    - Some instructions set only a subset of the possible values

- "Branch On Condition" instructions let you select alternate execution paths
  - Basic forms are BC (RX-type) and BCR (RR-type)
- Many other branch types extend these basic forms

- If the branch condition (defined on next slide) is *not* met:
  - Continue execution with the next sequential instruction
- If the branch condition *is* met:
  - For the BC (RX-type) instruction, the branch address is the Effective Address
  - For the BCR (RR-type) instruction, the branch address is the address in GR R_2
    - But if the R_2 digit is zero, no branch occurs and execution continues with the next sequential instruction
- The Instruction Address (IA) in the PSW is replaced by the branch address
  - So the next instruction to be fetched is at the branch address

- The branch condition is determined by testing a bit in the M_1 mask field of the instruction:



- The current value of the CC selects a bit in the M_1 mask;
  - If the bit is 0, the branch condition is *not* met
  - If the bit is 1, the branch condition *is* met

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

```
BCR   9,4          M1 = B'1001'  Branch if CC = 0,3
BC    7,4(8,2)     M1 = B'0111'  Branch if CC = 1,2,3
```


1. Branch to **XX** if the CC is zero.

```
BC    8,XX          M1 = B'1000'
```

2. Branch to **XX** if the CC is not 0.

```
BC    7,XX          M1 = B'0111'
```

3. Always 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 1, the CC value must match a 1-bit in the mask; this is called an *unconditional branch*

4. Branch to **XX** if the CC is 1 or 3.

```
BC    5,XX          M1 = B'0101'
```


No-Operation Instructions

- It's useful to have ("no-operation") instructions that do nothing
  - Often used to align other instructions on a specified boundary (see slide 20)
- These instructions never branch, never change the CC:

```
BC    0,x
BCR   0,any
```

- The Assembler provides special "extended mnemonics" for them:

```
NOP   S2     is equivalent to     BC    0,S2
NOPR  R2     is equivalent to     BCR   0,R2
```

- Some no-operation instructions have special side-effects:

```
BCR   15,0     ("branch always nowhere")
```

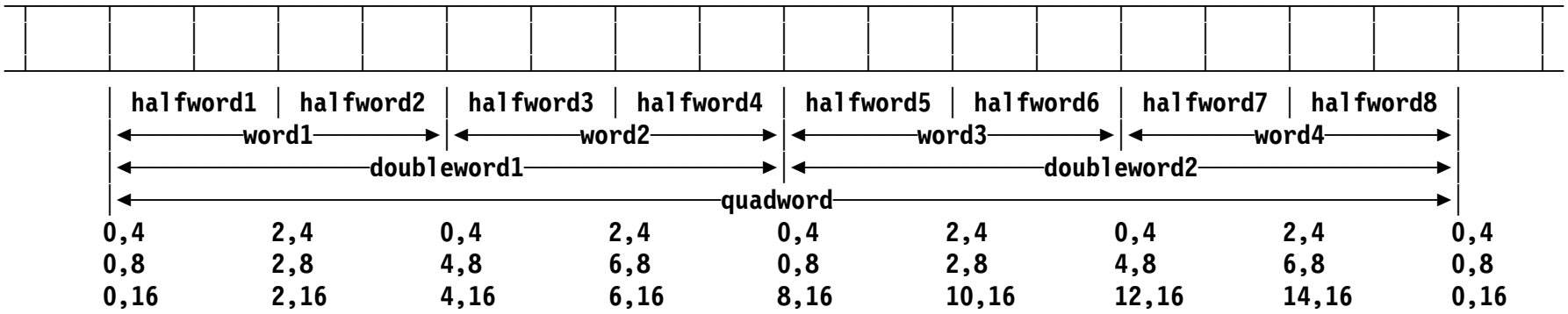
  causes internal overlaps of fetch/decode/execute phases to slow
  - Sometimes helps with problem diagnosis or potential memory conflicts

- Some instructions must be aligned on a specific boundary
  - Such as a BASR 14,15 instruction on a halfword boundary before a fullword
- Use the CNOP ("Conditional No-Operation") instruction to request the desired alignment

```
CNOP  boundary,width
```

  - **width** is either 4, 8, or 16
  - **boundary** is an even number such that **boundary < width**

```
CNOP  2,4      Align to the next halfword before a fullword
CNOP  0,8      Align to the next doubleword boundary
```



- Don't memorize mask-bit positions; use "Extended Mnemonics"

RX Mnemonic	RR Mnemonic	Mask	Meaning
B	BR	15	Unconditional Branch
BNO	BNOR	14	Branch if Not Ones or No Overflow
BNH	BNHR	13	Branch if Not High
BNP	BNPR	13	Branch if Not Plus
BNL	BNLR	11	Branch if Not Low
BNM	BNMR	11	Branch if Not Minus or Not Mixed
BE	BER	8	Branch if Equal
BZ	BZR	8	Branch if Zero(s)
BNE	BNER	7	Branch if Not Equal
BNZ	BNZR	7	Branch if Not Zero
BL	BLR	4	Branch if Low
BM	BMR	4	Branch if Minus, or if Mixed
BH	BHR	2	Branch if High
BP	BPR	2	Branch if Plus
BO	BOR	1	Branch if Ones, or if Overflow
NOP	NOPR	0	No Operation

- This section describes instructions for 2′s complement addition, subtraction, and comparison of many operand types
  - Between general registers
  - Between general registers and memory
  - Arithmetic and logical operations
  - Halfword, word, and doubleword operands
  - Mixed-length operands
    - If a source operand in a register or in memory is used as in an instruction whose target register is longer than the operand, the operand is extended internally:
      - arithmetic operands are sign-extended
      - logical operands are extended with zeros.
  - Add with carry, subtract with borrow
- Considerable symmetry among related instruction groups

- Instructions with 32-bit operands

Mnem	Function	Mnem	Function
AH	Add halfword from memory	SH	Subtract halfword from memory
A	Add word from memory	S	Subtract word from memory
AR	Add word from c(GR R_2)	SR	Subtract word from c(GR R_2)

- Instructions with 64-bit operands

Mnem	Function	Mnem	Function
AG	Add doubleword from memory	SG	Subtract doubleword from memory
AGR	Add doubleword from c(GG R_2)	SGR	Subtract doubleword from c(GG R_2)

  - Remember: GG is an abbreviation for “64-bit general register”

Signed-Arithmetic Operations With 32- or 64-Bit Registers 24

- CC settings for instructions with 32- or 64-bit results:

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) c(GG R_1) = c(GG R_1) ± c(GG R_2) c(GG R_1) = c(GG R_1) ± c(doubleword in memory)	0: Result is zero; no overflow 1: Result is < zero; no overflow 2: Result is > zero; no overflow 3: Result has overflowed

```
L     1,=F'2147483647'      2**31-1
A     1,=F'1'                          CC=3 (overflow)

L     2,=F'2147483647'      2**31-1
S     2,=F'1'                          CC=2 (positive)

L     3,=F'-2147483648'     -2**31
S     3,=F'1'                          CC=3 (overflow)

LG    4,=X'70000000 00000000'
AGR   4,4                              CC=3 (overflow)

LG    5,=X'50000000 00000000'
SG    5,=X'60000000 00000000'          CC=1 (negative)
```


- Comparisons with 32-bit operands

CH, CHY	Compare c(GR R_1) to halfword in memory
C, CY	Compare c(GR R_1) to word in memory
CR	Compare c(GR R_1) to word in c(GR R_2)

- Comparisons with 64-bit operands

CG	Compare c(GG R_1) to doubleword in memory
CGR	Compare c(GG R_1) to doubleword in c(GG R_2)

- Condition Code settings:

CC	Meaning
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

  - The CC cannot be set to 3 as a result of a compare instruction

Logical arithmetic produces bitwise identical results as the equivalent arithmetic operation; only the CC settings are different

- Instructions with 32- and 64-bit operands

Mnem	Function	Mnem	Function
AL	Add word from memory	SL	Subtract word from memory
ALR	Add word from c(GR R_2)	SLR	Subtract word from c(GR R_2)
ALG	Add doubleword from memory	SLG	Subtract doubleword from memory
ALGR	Add doubleword from c(GG R_2)	SLGR	Subtract doubleword from c(GG R_2)

- The CC settings:

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) c(GG R_1) = c(GG R_1) ± c(GG R_2) c(GG R_1) = c(GG R_1) ± c(doubleword in memory)	0: Zero result, no carry (Note) 1: Nonzero result, no carry 2: Zero result, carry 3: Nonzero result, carry
Note: CC0 cannot occur for logical subtraction	

- Instructions with 32- and 64-bit operands

Mnem	Function	Mnem	Function
ALC	Add word from memory	SLB	Subtract word from memory
ACLR	Add word from c(GR R_2)	SLBR	Subtract word from c(GR R_2)
ALCG	Add doubleword from memory	SLBG	Subtract doubleword from memory
ALCGR	Add doubleword from c(GG R_2)	SLBGR	Subtract doubleword from c(GG R_2)

- Each operation depends on the CC setting of a *preceding* instruction
  - Example: add and subtract pairs of 32-bit operands representing signed 64-bit integers

```
LM    0,1,A        Load A in register pair
AL    1,B+4        Logically 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

LM    0,1,A        Get first operand
SL    1,B+4        Logically subtract low-order parts
SLB   0,B          Subtract high-order parts with borrow
STM   0,1,Diff     Store 64-bit difference
```

- These instructions are especially useful for multi-precision arithmetic

Operations With Mixed 64-Bit and 32-Bit Operands 28

- The 32-bit operand is sign-extended internally to 64 bits before the operation
  - With sign bits for arithmetic instructions
  - With zero bits for logical instructions
- For these instructions, the first operand is in 64-bit register GG R_1

Mnem	Function	Mnem	Function
AGF	Add word from memory	SGF	Subtract word from memory
AGFR	Add word from c(GR R_2)	SGFR	Subtract word from c(GR R_2)
CGF	Compare to word from memory	CGFR	Compare to word from c(GR R_2)
ALGF	Logical add word from memory	SLGF	Logical subtract word from memory
ALGFR	Logical add word from c(GR R_2)	SLGFR	Logical subtract word from c(GR R_2)

- These instructions can simplify programs with mixed-length operands

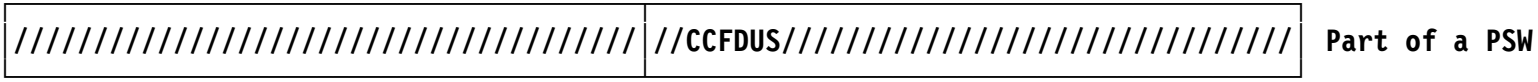
- Comparisons with 32- and 64-bit operands, and bytes in memory

Mnemonic	Function
CL	Logical compare c(GR R_1) to word from memory
CLR	Logical compare c(GR R_1) to word from c(GR R_2)
CLG	Logical compare c(GG R_1) to doubleword from memory
CLGR	Logical compare c(GG R_1) to doubleword from c(GG R_2)
CLGF	Logical compare c(GG R_1) to zero-extended word from memory
CLGFR	Logical compare c(GG R_2) to zero-extended word from c(GR R_2)
CLM, CLMY	Logical compare bytes from *low* half of c(GG R_1) to bytes in memory
CLMH	Logical compare bytes from *high* half of c(GG R_1) to bytes in memory

- The CC settings are the same as for other logical comparisons

- These instructions retrieve/set the PSW's CC and Program Mask (PM)

```
IPM   R1          Insert CC and Program Mask into GR R1
SPM   R1          Set CC and Program Mask from GR R1
```



  - "**CC**" represents the two bits of the CC. "**FDUS**" represents the four individual bits of the PM; they control whether an exception will (bit=1) or will not (bit=0) cause a program interruption:

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

```
SR    0,0         Set GR0 to zero
SPM   0           Set CC and Program Mask bits to zero
```


- These instructions move bits left and right in 32-bit GRs or GR pairs

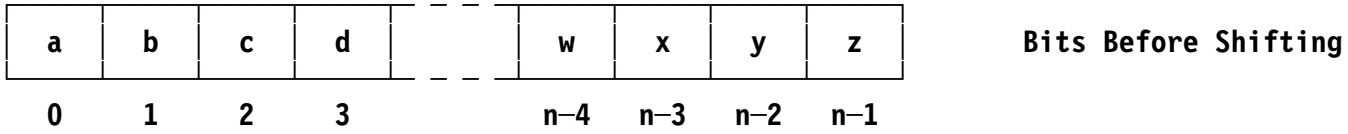
Mnem	Action	Mnem	Action
SLL	Shift left logical	SRL	Shift right logical
RLL	Rotate loft logical		
SLA	Shift left arithmetic	SRA	Shift right arithmetic
SLDL	Shift left double logical	SRDL	Shift right double logical
SLDA	Shift left double arithmetic	SRDA	Shift right double arithmetic

- These instructions move bits left and right in single 64-bit GRs

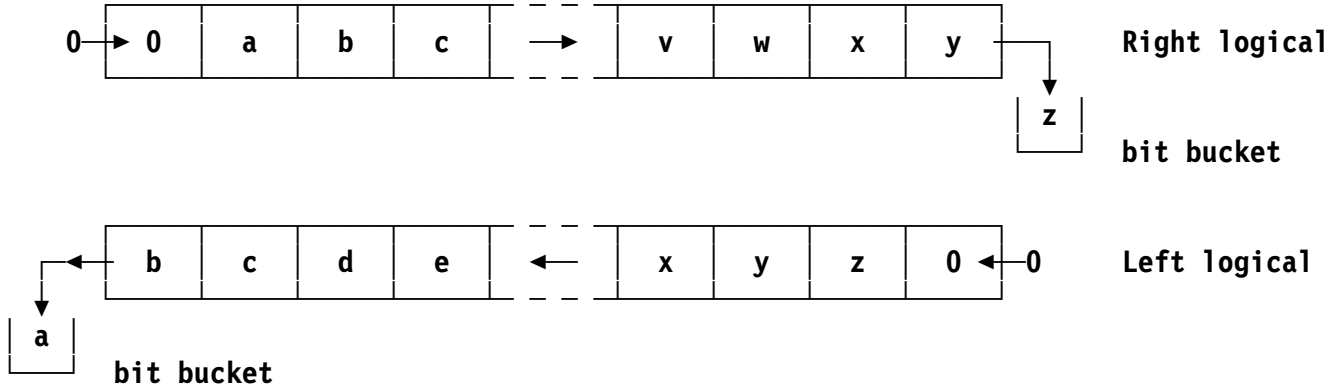
Mnem	Action	Mnem	Action
SLLG	Shift left logical	SRLG	Shift right logical
RLLG	Rotate loft logical		
SLAG	Shift left arithmetic	SRAG	Shift right arithmetic

- Shift amounts: the low-order 6 bits of the Effective Address
  - So shifts are limited to moving at most 63 bit posiions

- Assume an **n**-bit register looks like this:

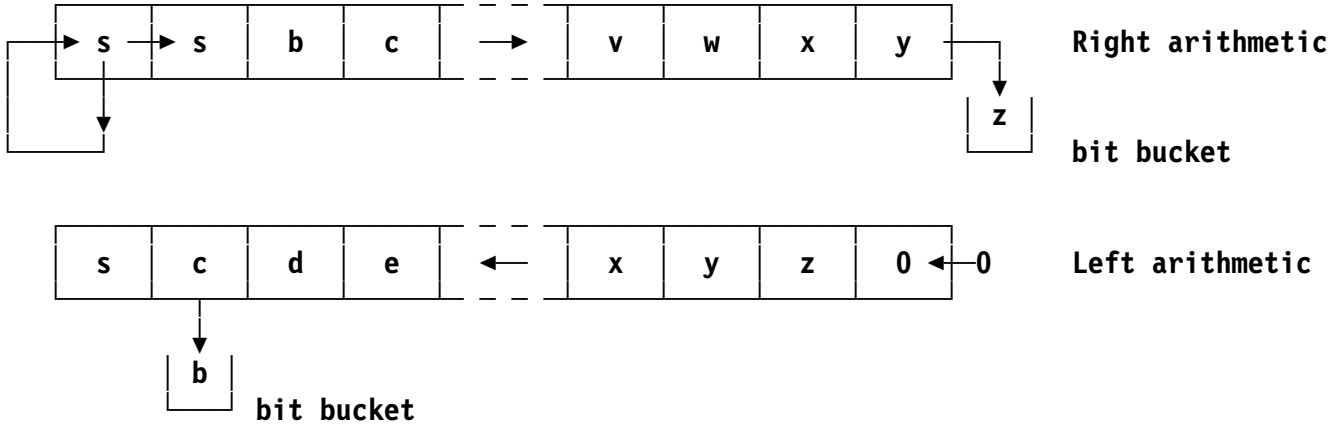


- A *unit shift* moves bits left or right by one position
- Logical shifts:



  - The Condition Code is unchanged

- Arithmetic shifts (“s” = sign bit):



  - The Condition Code is set

- For left shifts: if a lost bit differs from the sign bit (s ≠ b), an overflow exception is indicated

- For SLL and SRL: operands are $R_1,D_2(B_2)$

```
L     7,=X'87654321'     Initial contents of GR7
SLL   7,5(0)             Results: c(GR7) = X'ECA86420'
L     6,=X'87654321'     Initial contents of GR6
SRL   6,5(0)             Results: c(GR6) = X'043B2A19'
```

- For SLLG and SRLG: operands are $R_1,R_3,D_2(B_2)$
  - The operand in GG R_3 is shifted, the result is placed in GG R_1

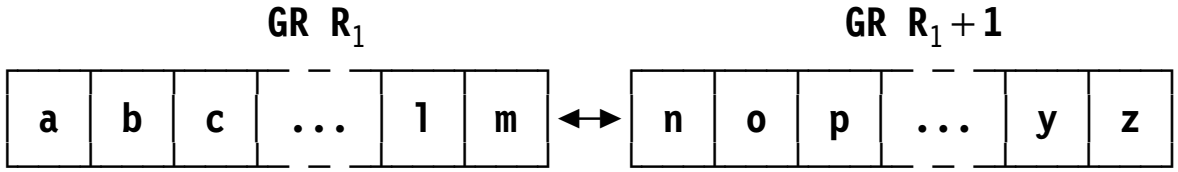
```
LG    1,=XL8'123456789ABCDEF0'     c(GG1) initialized
SLLG  0,1,9                 c(GG0) = X'68ACF135 79BDE000'

LG    1,=XL8'123456789ABCDEF0'     c(GG1) initialized
SRLG  0,1,9                 c(GG0) = X'00091A2B 3C4D5E6F'
```

    - Because the R_1 and R_3 operands differ, c(GG1) remains unchanged

- Exercise: Verify the example results

- Double-length logical *and* arithmetic shifts use an even-odd GR pair




```
LM    0,1,=X'123456789ABCDEF0'      GR(0,1) initialized
SLDL  0,9       c(GR0) = X'68ACF135', c(GR1) = X'798DE000'

LM    2,3,=X'123456789ABCDEF0'      GR(2,3) initialized
SRDL  2,9       c(GR0) = X'00091A28', c(GR1) = X'3C4D5E6F'
```

  - Exercise: Verify the results of the SLDL and SRDL examples.

- Example: is the number in GR6 a multiple of 32 (2^5)?

```
SR    7,7           Set c(Gr7) to zero
SRDL  6,5           Move rightmost 5 bits into GR7
LTR   7,7           Are those bits zero?
BZ    Yes_It_Is     If the bits are zero, it's a multiple
B     Sorry_No      Sorry, it's not a multiple of 32
```


- Arithmetic shifts always set the CC:

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

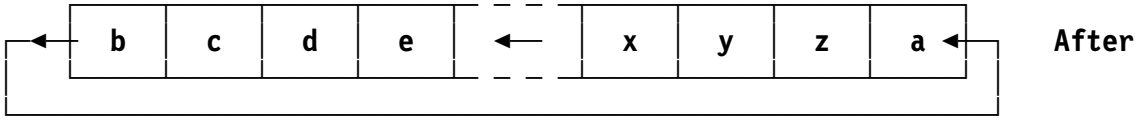
- Examples:

```
L     6,=X'87654321'
SRA   6,5                 c(GR6) = X'FC3B2A19', CC=1

L     7,=X'87654321'
SLA   7,5                 c(GR7) = X'ECA86420', CC=3 (oveflow)
```

  - Remember: for left shifts, if a lost bit differs from the sign bit (s ≠ b), an overflow exception is indicated

A Rotating shift looks like this (compare slide 32)



- Like SLLG and SRLG, RLL and RLLG have three operands: $R_1,R_3,D_2(B_2)$

```
L     0,=A(X'56789ABC')   Load initial data into GR0
RLL   1,0,10              Rotate 10 bits, result in GR1
```

  - Then c(GR1) = X'E26AF159'

```
LG    0,=AD(X'56789ABCDEF01234')   Initialize GG0
RLLG  1,0,10                       Rotate 10 bits, result in GG1
```

  - Then c(GG1) = X'E26AF37BC048D159'

- None of the rotating-shift instructions changes the CC

- Some shift amounts must be determined at execution time
- Solution: put the shift amount in the B_2 register

```
L     9,ShiftAmt          Shift amount calculated previously
L     0,Data              Get data to be shifted
SLL   0,0(9)              Shift left by calculated amount
```

  - Remember: only the low-order 6 bits of the Effective Address are used for the shift count

- Example: calculate 2^N, where $0 \leq N < 31$

```
L     1,N                 Get small integer N
L     0,=F'1'             Put a 1–bit at right end of GR0 (2⁰)
SLL   0,0(1)              Leave 2^N in GR0
```

  - Exercise: What will happen if $N \geq 31$?

Bit-Length Constants

An extended form of length modifier lets you specify lengths in bits: put a period (.) after the modifier **L**

```
DC  FL3'8'      and      DC  FL.24'8'    are equivalent
(byte length)            (bit length)
```

- 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'
```

- Bit-length constants are useful for tightly packed data

- Multiplication notation:

```
   Multiplicand      First operand
 × Multiplier        Second operand
   Product           Single or double-length result
```

  - Products can be as long as the sum of the operand lengths: 456 × 567 = 258552, or as short as the longer operand: 456 × 2 = 912

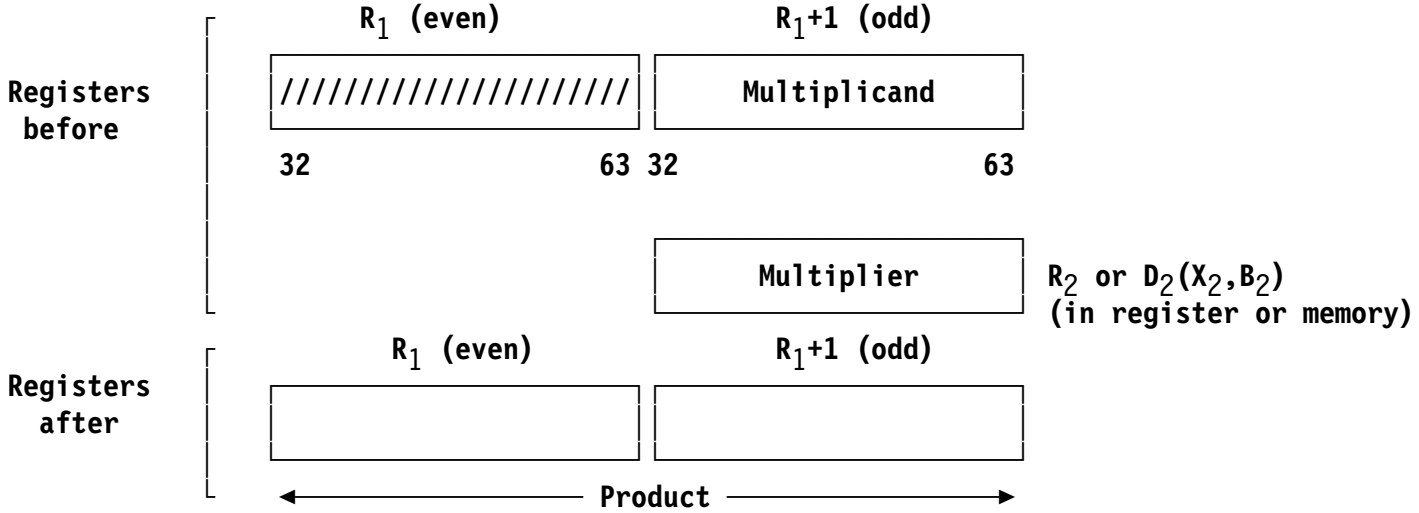
- Multiplying single-length operands usually requires a double-length register pair
- Division notation:

```
          Quotient
Divisor ) Dividend      Dividend = first operand
           - - - -      Divisor = second operand
          Remainder
```

- Division by a single-length divisor usually requires a double-length dividend, producing single-length quotient and remainder
- Multiply and divide instructions do not change the CC

- For 64-bit signed products of 32-bit operands:

Mnem	Instruction	Mnem	Instruction
M	Multiply (32+32←32×32)	MR	Multiply Register (32+32←32×32)



- Example(1): Square the number in GR1

```
MR    0,1          c(GR0,GR1) = c(GR1)*c(GR1)
```

- Example(2): Square the number in GR0

```
LR    1,0          Copy c(GR0) to GR1 (odd register of pair)
MR    0,0          Multiply c(GR0) by c(GR1)
```


- When multiplying small values, you can use a single register:

Mnem	Instruction	Mnem	Instruction
MH	Multiply Halfword (32←32×16)		
MS, MSY	Multiply Single (32←32×32)	MSR	Multiply Single Register (32←32×32)
MSG	Multiply Single (64←64×64)	MSGR	Multiply Single Register (64←64×64)
MSGF	Multiply Single (64←64×32)	MSGFR	Multiply Single Register (64←64×32)

- Examples

```
MH     5,=H'100'          c(GR5)*100

LH     1,N                Load c(N) into GR1
MSR    1,1                c(GR1)*C(GR1) = N2

LG     1,=FD'12345678'    c(GG1) = 12345678
MSG    1,=FD'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
```


- For 64-bit products of two 32-bit unsigned operands

Mnem	Instruction	Mnem	Instruction
ML	Multiply Logical (32+32←32×32)	MLR	Multiply Logical Register (32+32←32×32)

- For 128-bit products of two 64-bit unsigned operands

Mnem	Instruction	Mnem	Instruction
MLG	Multiply Logical (64+64←64×64)	MLGR	Multiply Logical Register (64+64←64×64)

- Examples:

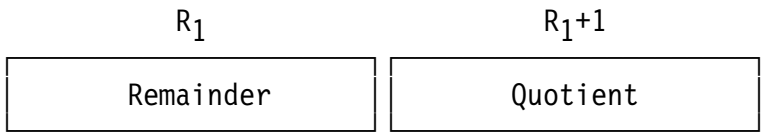
```
*  Logical multiplication: (2**32-1)*(2**32-1) = 18446744065119617025
        L     1,=F'-1'            c(GR1) = X'FFFFFFFF'
        MLR   0,1                 c(GR0,GR1) = X'FFFFFFFE 00000001'

        LG    1,=FD'74296604373'  c(GG1) = 74296604373
        MLG   0,=FD'9876543210'   c(GG0,GG1) = 733793623446209457330
```


- Dividing a 2n-digit dividend by an n-digit divisor may not produce an n-digit quotient:

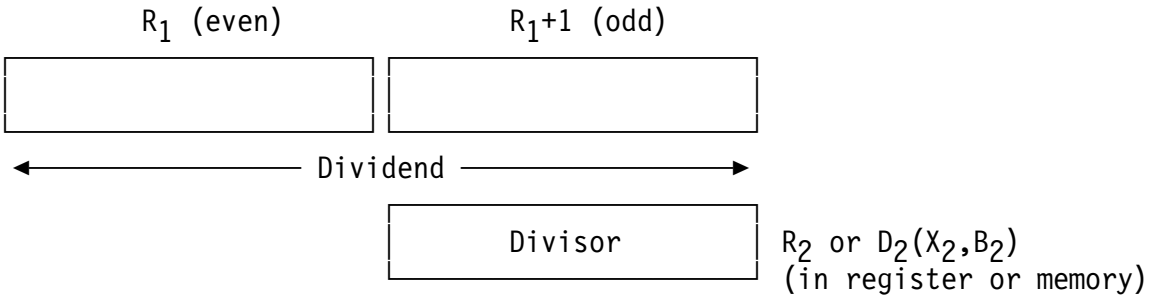
  987*867 = 855729; 855729/123 = 6957, remainder 18

- If the attempted quotient is too big for a register, or a divisor is zero. a Fixed Point Divide Interruption always occurs (it can't be masked off)
- All binary division instructions require an even-odd register pair
  - Dividends occupy either an even-odd pair or an odd register
- Quotient and remainder of a successful division:

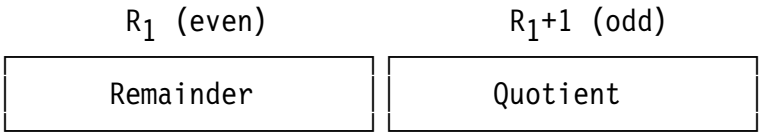


- These instructions support signed-operand division using a double-length dividend:

Mnem	Instruction	Mnem	Instruction
D	Divide (32,32←32+32÷32)	DR	Divide Register (32,32←32+32÷32)



  - The results are in an even-odd register pair

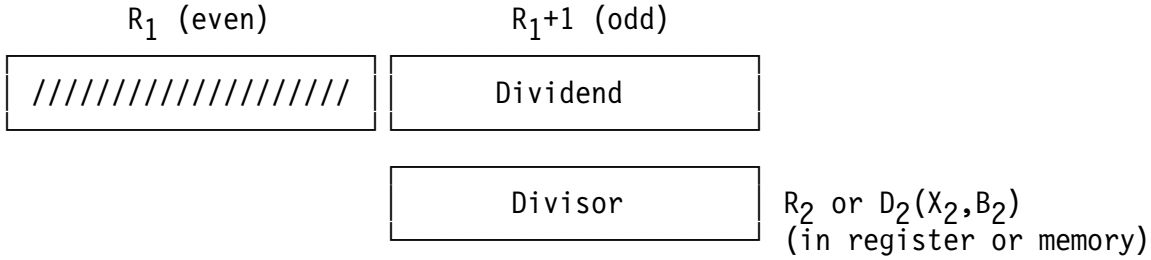


- Example

```
L     2,=F'-14352'      Dividend in GR2
SRDA  2,32              Create double-length dividend
D     2,=F'17'          c(GR2) = -4. c(GR3) = -844
```


- The Divide Single instructions use a single-length dividend in an odd register; the results are as above.

Mnem	Instruction	Mnem	Instruction
DSG	Divide Single (64,64←64÷ 64)	DSGR	Divide Single Register (64,64←64÷ 64)
DSGF	Divide Single (64,64←64÷ 32)	DSGFR	Divide Single Register (64,64←64÷ 32)



- Examples:

```
          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)
```


- These instructions consider *all* operands unsigned:

Mnem	Instruction	Mnem	Instruction
DL	Divide Logical (32,32←32+32÷32)	DLR	Divide Logical Register (32,32←32+32÷32)
DLG	Divide Logical (64,64←64+64÷64)	DLGR	Divide Logical Register (64,64←64+64÷64)

- The dividend is always double-length
- Example:

```
          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'
*                                 Quotient and remainder = X'FFFFFFFE'
```

- Only unsigned operands are used for dividing 128-bit dividends

Summary of Multiply and Divide Instructions 48

Function	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

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

- System z instructions perform three logical operations: AND, OR, and Exclusive OR ("XOR")
- Each operates strictly between corresponding pairs of bits:

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

  - Neighboring bits are unaffected, and do not participate
- The instructions here operate only on registers;
  others (later!) operate on single memory bytes or strings of bytes

- Instructions with 32-bit operands:

Mnem	Instruction	Mnem	Instruction
N, NY	AND (32)	NR	AND Register (32)
O, OY	OR (32)	OR	OR Register (32)
X, XY	Exclusive OR (32)	XR	Exclusive OR Register (32)

- Instructions with 64-bit operands:

Mnem	Instruction	Mnem	Instruction
NG	AND (64)	NGR	AND Register (64)
OG	OR (64)	OGR	OR Register (64)
XG	Exclusive OR (64)	XGR	Exclusive OR Register (64)

- Each instruction sets the Condition Code:

Operation	CC setting
AND OR XOR	**0:** all result bits are zero **1:** result bits are not all zero

- Consider each operation, using identical operands:

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

1. Exchange the contents of two registers:

```
XR    1,2
XR    2,1
XR    1,2
```

2. Turn off the rightmost 1-bit of a positive number X:

   Y = X AND (X–1)

   - Example:

```
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'
```

3. Isolate the rightmost 1-bit of a word

   Y = X AND (–X)

   - Example:

```
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'
```


Chapter VI: Addressing, Immediate Operands, and Loops 1

This chapter describes three useful and important topics:

- Section 20 discusses ways the CPU can generate Effective Addresses, and how those addresses depend on the current addressing mode
- Section 21 introduces instructions with *immediate* operands, where one of the operands of the instruction is contained in the instruction itself
- Section 22 reviews instructions that help you manage loops: iterative execution of a block of instructions that perform some repeated action

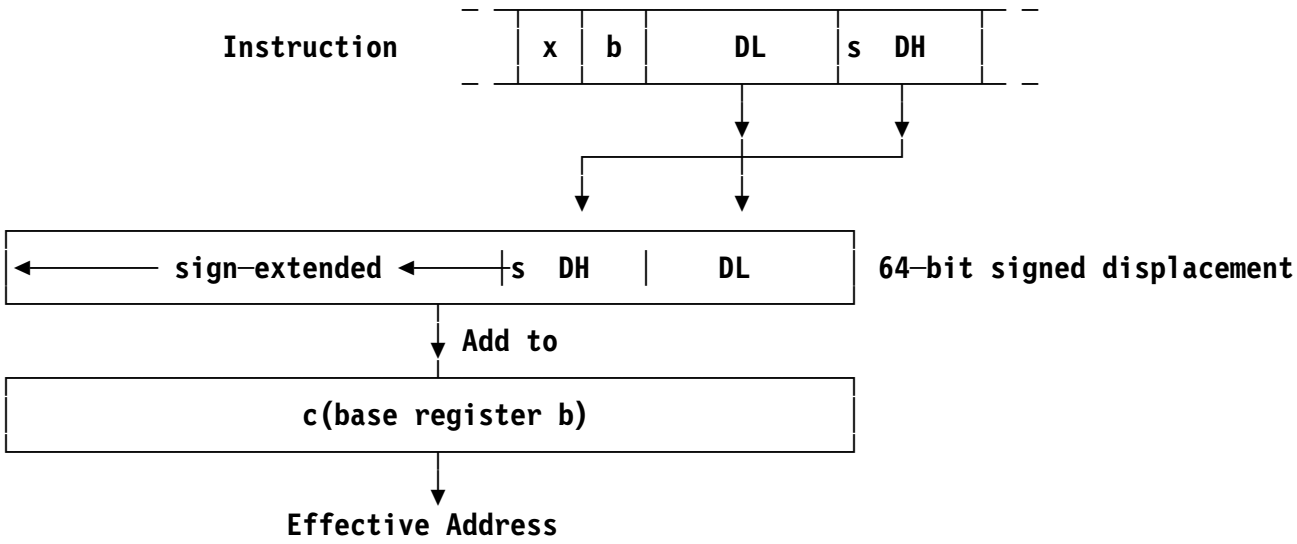
- System z supports three types of address generation:
  1. base-displacement with unsigned 12-bit displacements
     - This was described in Section 5
  2. base-displacement with signed 20-bit displacements
  3. relative-immediate.
- ... and three addressing *modes*, which define the number of rightmost bits of an Effective Address that are actually used for addressing:

  At any given moment, only one of 24-, 31-, or 64-bit modes is active

- Instructions with signed 20-bit displacements have this format:

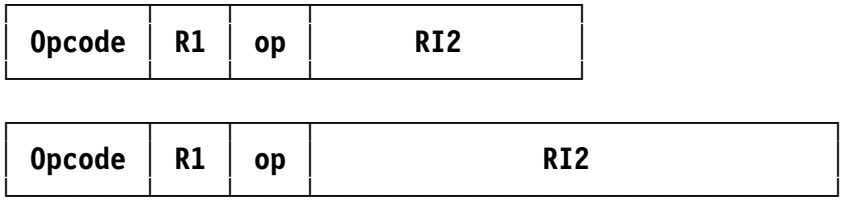
opcode	R_1	X_2	B_2	DL_2	DH_2	opcode

- Address generation first creates a *signed* displacement:

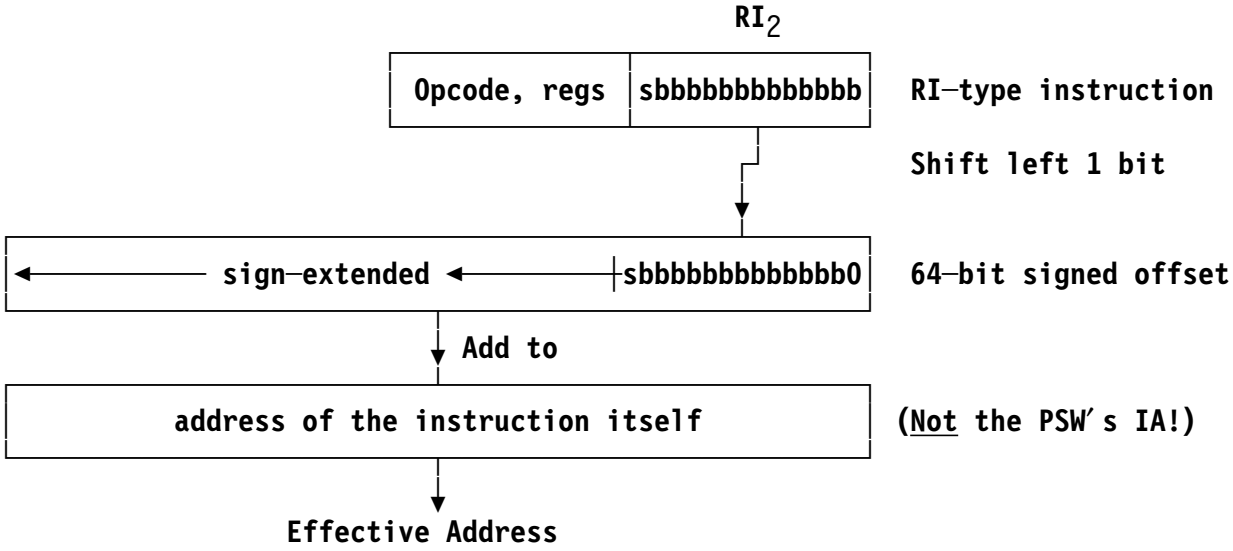


- Addressing range is ±512K (vs. 4K for unsigned 12-bit displacements)

- Relative-immediate instructions have two basic formats:

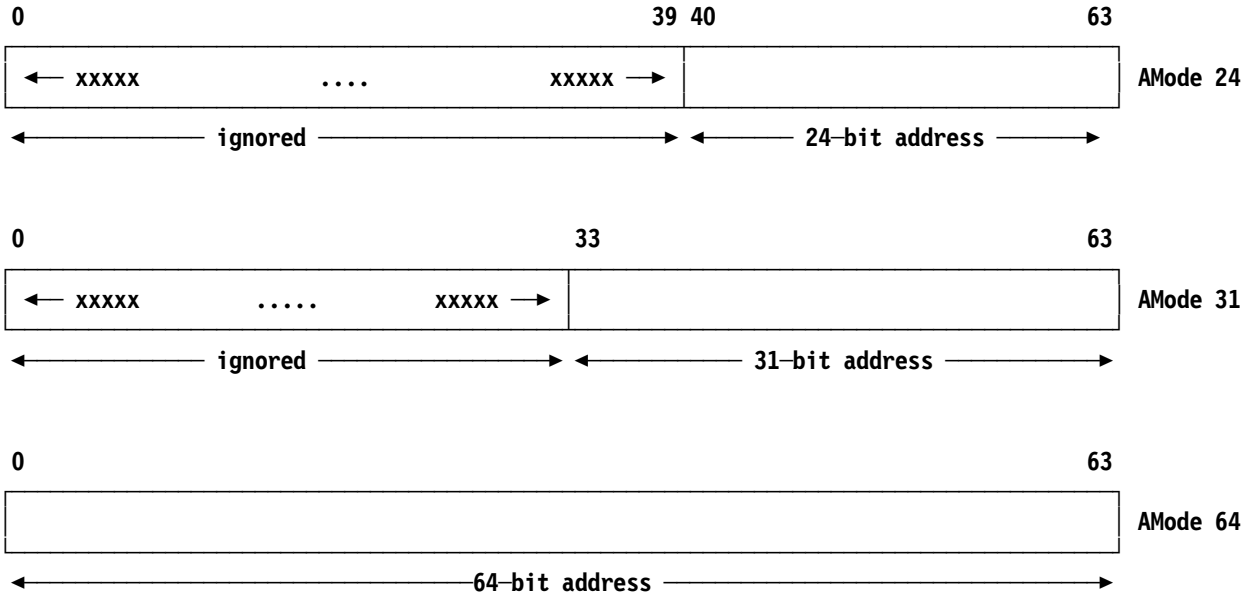


- Address generation involves *signed* offsets:



- Addressing range is ±64KB (16-bit offset) or ±4GB (32-bit offset)

- "Addressing Mode" is often abbreviated "AMode"
- Determines which bits of an Effective Address are used for addressing:



- These instructions put the Effective Address in the first operand register

Mnem	Instruction	Mnem	Instruction
LA, LAY	Load Address	LARL	Load Address Relative Long

- Small constants can be put in a GR using LA, LAY:

```
LA    0,n(0,0)        0 ≤ n ≤ 4095
LAY   0,n(0,0)     –2^19 ≤ n ≤ 2^19–1
```

- These instructions are **modal**: the results depend on the AMode

```
LAY   0,–1            Put –1 in register 0
```

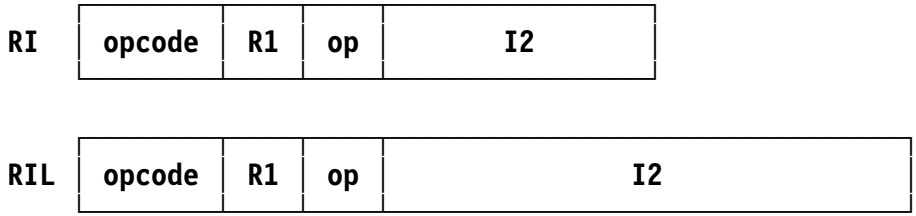
  - 24-bit mode: result in GR0 is X'00FFFFFF'
  - 31-bit mode: result in GR0 is X'7FFFFFFF'
  - 64-bit mode: result in GG0 is X'FFFFFFFFFFFFFFFF'

- LARL always creates a memory address (relative to LARL)

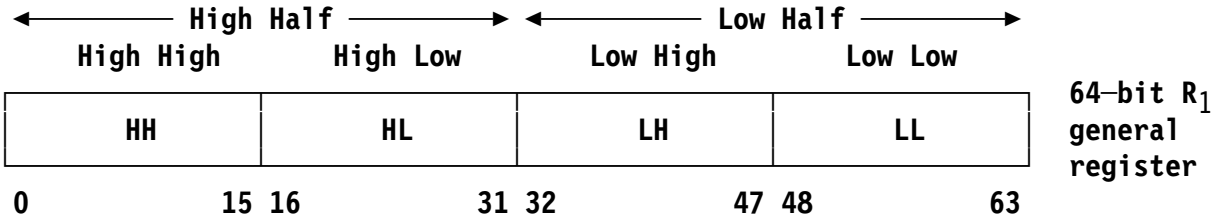
- This table summarizes the three Load Address instructions:

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.

- "Immediate" operands are part of the instruction itself
  - SI-type was described in Section 4.2 (more about them in Section 23)
    - The other operand is in memory
  - RI-, RIL-types involve a register operand



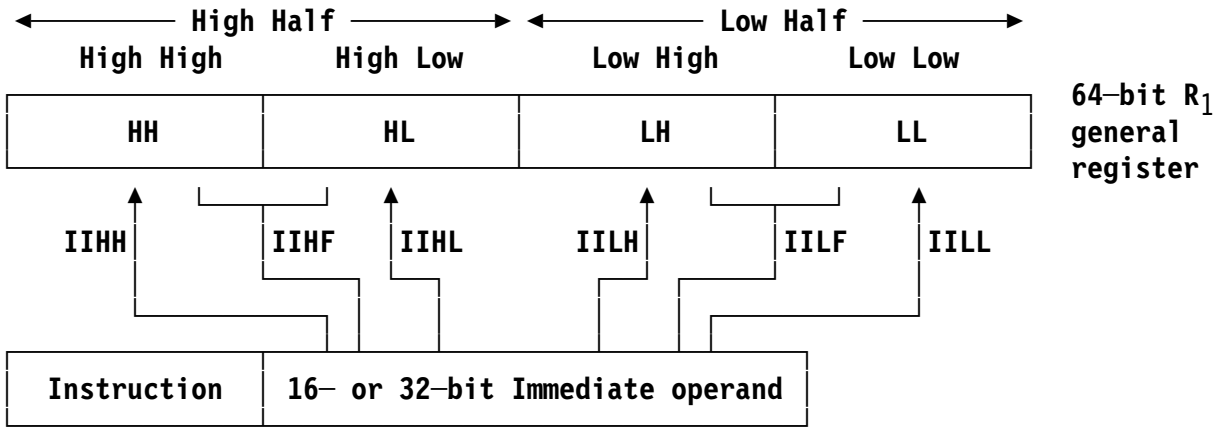
- Some instructions affect an entire register, some only parts:



  - **H** = High Half; **HL** = High Half's Low Half, etc.

- These instructions insert an operand into part of a register without changing *any other* part

Mnem	Instruction	Mnem	Instruction
IIHF	Insert Logical Immediate (high) (64←32)	IILF	Insert Logical Immediate (low) (64←32)
IIHH	Insert Logical Immediate (high high) (64←16)	IIHL	Insert Logical Immediate (high low) (64←16)
IILH	Insert Logical Immediate (low high) (64←16)	IILL	Insert Logical Immediate (low low) (64←16)



- The three arithmetic loads sign-extend the immediate operand:

Mnem	Instruction	Mnem	Instruction
LHI	Load Halfword Immediate (32←16)	LGHI	Load Halfword Immediate (64←16)
LGFI	Load Immediate (64←32)		

- These logical loads zero *all other* parts of the register:

Mnem	Instruction	Mnem	Instruction
LLIHF	Load Logical Immediate (high) (64←32)	LLILF	Load Logical Immediate (low) (64←32)
LLIHH	Load Logical Immediate (high high) (64←16)	LLIHL	Load Logical Immediate (high low) (64←16)
LLILH	Load Logical Immediate (low high) (64←16)	LLILL	Load Logical Immediate (low low) (64←16)

  - Their operation is similar to the Insert-Immediate instructions (which don't zero other parts of the register)

- Arithmetic and logical add and subtract instructions:

Mnem	Instruction	Mnem	Instruction
AHI	Add Halfword Immediate (32←16)	AGHI	Add Halfword Immediate (64←16)
AFI	Add Immediate (32)	AGFI	Add Immediate (64←32)
ALFI	Add Logical Immediate (32)	ALGFI	Add Logical Immediate (64←32)
SLFI	Subtract Logical Immediate (32)	SLGFI	Subtract Logical Immediate (64←32)

- Arithmetic and logical compare instructions:

Mnem	Instruction	Mnem	Instruction
CHI	Compare Halfword Immediate (32←16)	CGHI	Compare Halfword Immediate (64←16)
CFI	Compare Immediate (32)	CGFI	Compare Immediate (64←32)
CLFI	Compare Logical Immediate (32)	CLGFI	Compare Logical Immediate (64←32)

- Multiply instructions with an immediate operand:

Mnem	Instruction	Mnem	Instruction
MHI	Multiply Halfword Immediate (32←16)	MGHI	Multiply Halfword Immediate (64←16)

- These instructions perform AND, OR, and XOR of an immediate operand and a register, and leave the result in the register

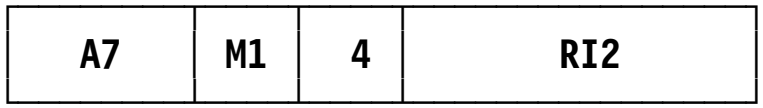
Mnem	Instruction	Mnem	Instruction
NIHF	AND Immediate (high) (64←32)	NILF	AND Immediate (low) (64←32)
NIHH	AND Immediate (high high) (64←16)	NIHL	AND Immediate (high low) (64←16)
NILH	AND Immediate (low high) (64←16)	NILL	AND Immediate (low low) (64←16)
OIHF	OR Immediate (high) (64←32)	OILF	OR Immediate (low) (64←32)
OIHH	OR Immediate (high high) (64←16)	OIHL	OR Immediate (high low) (64←16)
OILH	OR Immediate (low high) (64←16)	OILL	OR Immediate (low low) (64←16)
XIHF	XOR Immediate (high) (64←32)	XILF	XOR Immediate (low) (64←32)

- The instructions in this section can help in many ways:
  1. They eliminate the need to access storage
  2. They save the space those operands needed
  3. They can save base registers once needed for memory addressing

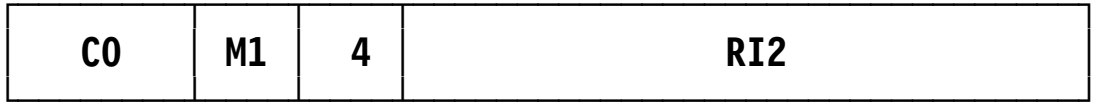
Operation	Operand 1	32 bits		64 bits	
	Operand 2	16 bits	32 bits	16 bits	32 bits
Arithmetic Add/Subtract		AHI	AFI	AGHI	AGFI
Logical Add/Subtract			ALFI SLFI		ALGFI SLGFI
Arithmetic Compare		CHI	CFI	CGHI	CGFI
Logical Compare			CLFI		CLGFI
Multiply		MHI		MGHI	

- This section describes three powerful types of branch instruction:
  - **Branch Relative on Condition**: these are similar to the familiar Branch on Condition instructions, but with a relative-immediate operand address
  - **Branch on Count**: these help control the execution of loops controlled by the number of iterations
  - **Branch on Index**: these powerful instructions can increment an index value, compare the sum to an end value, and determine whether or not to branch, all in a single instruction
- We will also examine some general styles of loop organization

- The BRC and BRCL instructions have these formats:



  - The branch target can be as far away as −65536 and +65534 bytes



  - The branch target can be more than 4 billion bytes away from the branch instruction, in either direction

  This means the offset of the branch target can be more than 4 billion bytes away from the RIL-type instruction, in either direction

- The greatest advantage of these branch instructions is that no base register is needed for addressing instructions
- Their extended mnemonics are described on slide 16

- The extended mnemonics are formed by adding the same suffixes as for based branch instructions to “BRC” and “BRCL”
  - To distinguish them from based branches, different prefixes are sometimes used: **J** (for “Jump”) and **JL** (for “Jump Long”). For example:

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 No Overflow
BRNH	JNH	BRNHL	JLNH	13	Branch if Not High
BRNP	JNP	BRNPL	JLNP	13	Branch if Not Plus
BRNL	JNL	BRNLL	JLNL	11	Branch if Not Low
BRNM	JNM	BRNML	JLNM	11	Branch if Not Minus
BRE	JE	BREL	JLE	8	Branch if Equal
⋮	⋮	⋮	⋮	⋮	⋮
BRP	JP	BRPL	JLP	2	Branch if Plus
BRO	JO	BROL	JLO	1	Branch if Overflow

- Programs often deal with tables of data
  - The program might scan the table from “top to bottom” stepping from one row to the next
    - This is called “sequential scanning of a one-dimensional array” (more in Section 40)
  - Example: Add the integers in a table:

```
         LHI   2,10              Count of numbers in the table
         XR    1,1               Set index to zero
         XR    0,0               Set sum to zero
Add      A     0,Table(1)        Add an integer from the table
         AHI   1,4               Increment index by number length
         AHI   2,-1              Reduce count by 1
         JNZ   Add               If not zero, repeat
         ST    0,Sum             Store the resulting sum
         - - -
Table    DC    F'1,2,3,4,5,6,7,8,9,10'
```

- We used c(GR1) as the “index” to reference each item in turn

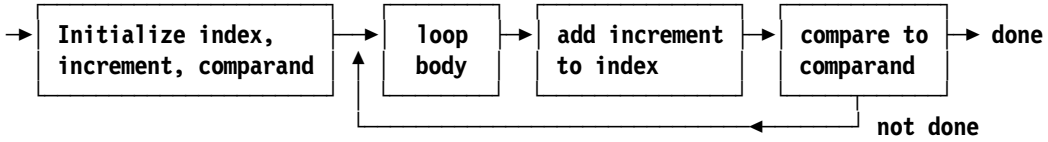
- Count-controlled loops are easily managed with these instructions

Mnem	Instruction	Mnem	Instruction
BCT	Branch on Count (32)	BCTR	Branch on Count Register (32)
BCTG	Branch on Count (64)	BCTGR	Branch on Count Register (64)
BRCT, JCT	Branch Relative on Count (32)	BRCTG, JCTG	Branch Relative on Count (64)

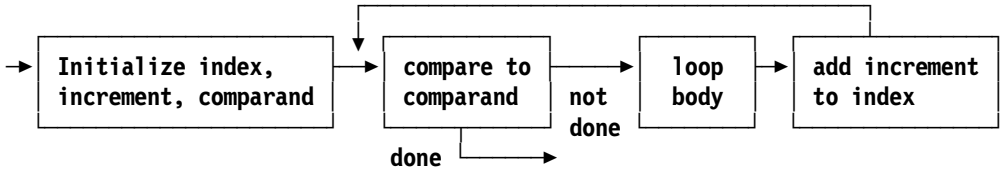
- Execution follows these steps:
  1. Reduce the number in the R_1 register by one
     - For BCTR and BCTGR: if the R_2 operand is zero, do nothing more
  2. If the result is zero, do not branch; fall through to the next sequential instruction
  3. If the result is zero, branch to the instruction at the Effective Address
- Example: add the numbers from 1 to 10 (in reverse order)

```
         XR    0,0          Clear GR0 for the sum
         LA    1,10         Number of values to add
Repeat   AR    0,1          Add a value to the sum
         BCT   1,Repeat     Reduce counter by 1, repeat if nonzero
         ST    0,Sum        Store the result for display
```

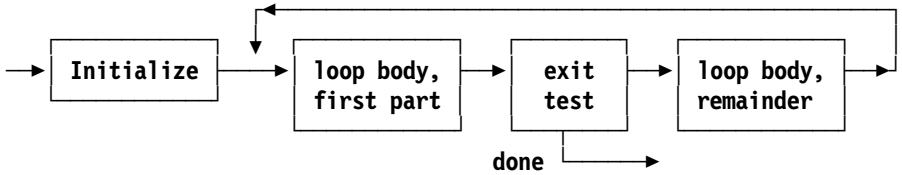

- There are many types of loop; some of the most common are:
  1. “Do-Until”



  2. “Do-While”

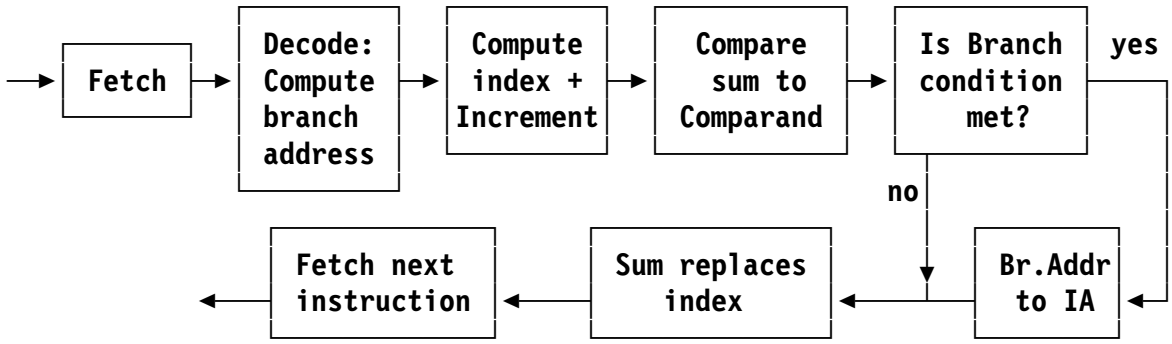


  3. A combination



- These instructions use 32- or 64-bit operands, and based or relative-immediate branch addresses; they combine incrementation, comparison, and branching in a single instruction

Mnem	Instruction	Mnem	Instruction
BXH	Branch on Index High (32)	BXHG	Branch on Index High (64)
BXLE	Branch on Low or Equal (32)	BXLEG	Branch on Index Low or Equal (64)
BRXH, JXH	Branch Relative on Index High (32)	BRXHG, JXHG	Branch Relative on Index High (64)
BRXLE, JXLE	Branch Relative on Low or Equal (32)	BRXLG, JXLEG	Branch Relative on Index Low or Equal (64)



- For all the “Branch on Index” instructions:
- The R_1 operand is the index; the increment is in R_3, and the comparand is in $R_3|1$ (R_3 value with low-order bit forced to 1)
- Examples
  1. Add the numbers in a table with a two-instruction loop

```
         XR    0,0              Set sum to zero
         XR    1,1              Set index to zero
         LM    2,3,=F'4,36'     Initialize increment, comparand
Add      A     0,Table(1)       Add an integer from the table
         BXLE  1,2,Add          Increment index, compare to 36
         _ _ _
Table    DC    F'1,2,3,4,5,6,7,8,9,10'
```

  2. Another way to do the same, adding successive index values

```
         XR    0,0              Clear sum to zero
         LHI   1,1              Initialize “index” to 1
         LM    2,3,=F'1,10'     Initialize increment and comparand
Add      AR    0,1              Add “index” to sum
         BXLE  1,2,Add          Repeat 10 times
```


- BXH is often used for indexing from “bottom to top”
- Examples
  1. Add the numbers in a table with a two-instruction loop

```
            SR    0,0              Clear sum to zero
            LHI   1,36             Initialize index to 36 (last element)
            L     3,=F'-4'         Identical increment and comparand
Add         A     0,Table(1)       Add an element of the table
            BXH   1,3,Add          Decrease index, compare to -4
            - - -
Table       DC    F'3,1,4,1,5,9,2,6,5,89'
```

  2. Calculate a table of cubes of the first 10 integers

```
            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
```


- BXH and BXLE can do some interesting things. Three examples:

  1. Branch to XXX if c(GR4) is ≤ 0

```
         XR    9,9              Set GR9 to zero
         BXLE  4,9,XXX          Branch to XXX if c(GR4) is <= 0
```

  and...!

```
         XR    9,9              Set GR9 to zero
         BXH   4,9,YYY          Branch to YYY if c(GR4) is > 0
```

  2. If c(GR2) > 0, branch to XXX after adding 1 to c(GR2)

```
         LHI   7,1              Initialize GR7 to +1
         BXH   2,7,XXX          Increment c(GR2), branch to XXX
```

  3. If c(GR4) = +1, then increment c(GR5) by 1 and branch to ZZZ if the sum doesn't overflow

```
         BXH   5,4,ZZZ
```


- These instructions are described in Section 22:

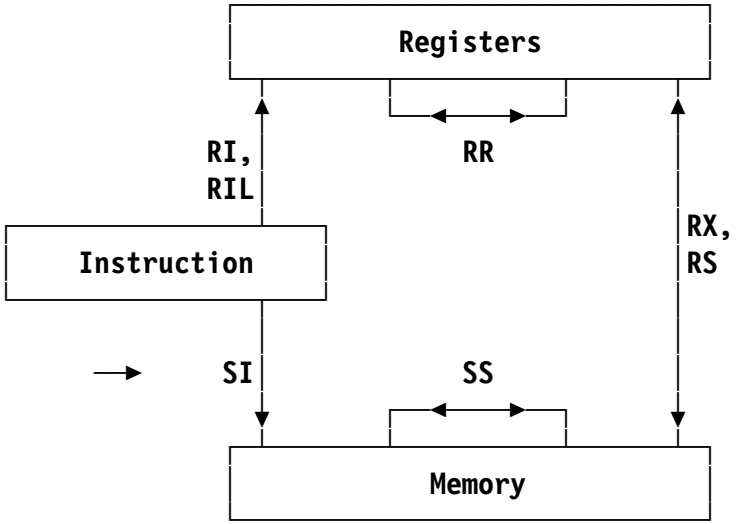
Operation	Relative-Immediate Operand Length	
	16 bits	32 bits
Branch on Condition (Relative)	BCR	BCRL

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

Bit and Character Data 1

- Section 23 describes new data types and related instructions:
  - Individual bits and bytes
  - Varying-length character strings
- Section 24 describes SS-type instructions in detail:
  - Frequently-used instructions handling large or variable numbers of bytes
  - The powerful “Execute” instructions
- Section 25 examines instructions that handle very long strings of bytes, and byte strings containing special characters
- Section 26 introduces other character representations and associated instructions, including:
  - The popular ASCII character set
  - Unicode, that can represent almost all known characters
  - Other multiple-byte characters

- Unlike RI- and RIL-type instructions, the target operand of SI-type instructions is a byte in memory.



- SI-type source operands are single bytes
  - For RI and RIL types, the source and target operands can have different lengths
    - The resulting register operand can be longer than the immediate (source) operand

- We'll discuss these instructions:

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

- The instructions have these SI- and SIY-type formats:

- Write the operand field as either **$D_1(B_1),I_2$** or **S_1,I_2**

- There are two Move Immediate instructions:

Mnem	Instruction	Mnem	Instruction
MVI	Move Immediate	MVIY	Move Immediate

- Each stores its I_2 operand byte at the Effective Address
- Examples:

```
MVI   X,0                Set the byte at X to all 0-bits
MVI   X,255              Set the byte at X to all 1-bits
MVI   X,C' '             Store EBCDIC blank at X

MVI   FlagByte,0         Set all flag bits to zero
MVI   CrrgCtrl,C'1'      Printer carriage control for new page
```


Mnem	Instruction	Mnem	Instruction
NI	AND Immediate	NIY	AND Immediate
OI	OR Immediate	OIY	OR Immediate
XI	XOR Immediate	XIY	XOR Immediate

- Each instruction sets the Condition Code:

Operation	CC setting
AND OR XOR	**0:** all result bits are zero **1:** result bits are not all zero

```
(a)          NI    X,0                Same as 'MVI X,0' except CC set to 0
             NI    X,B'11111101'      Sets bit 6 at X to 0
(b)          OI    X,255              Same as 'MVI X,255' except CC set to 1
             OI    X,B'00000010'      Sets bit 6 at X to 1
(c)          OI    LowerA,C' '        c(LowerA) now is C'A'
   LowerA    DC    C'a'               Initially, lower case letter 'a'
(c)          XI    X,B'0000010'       Inverts bit 6 at X
```


Mnem	Instruction	Mnem	Instruction
CLI	Compare Immediate	CLIY	Compare Immediate

- The first operand is compared logically to the second, to set the CC:

CC	Indication
0	Operand 1 = I_2
1	Operand 1 < I_2
2	Operand 1 > I_2

- Note: The *first* operand is the byte in memory at the Effective Address.

```
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
```


Mnem	Instruction	Mnem	Instruction
TM	Test Under Mask	TMY	Test Under Mask

- The 1-bits of the immediate I_2 operand indicate which corresponding bits of the first operand will be tested; the CC shows the result

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

```
TM    Num,X'80'          Test leftmost bit of a number
JO    Minus              Branch if a 1-bit, it's negative

TM    Num+L'Num-1,1      Test rightmost bit of a number
JZ    Even               Branch if low-order bit is zero

TM    BB,255             Test all eight bits
JM    Mixed              Branch if not all zeros or all ones
```


- It's better to *name* bit-data items than to use bit numbers or bit masks
- Discouraged techniques:

```
          OI    Person_X,128       Person has retired [Poor method]
or
          OI    Person_X,Bit1      Person has retired [Poor method]
```

- Better technique: name each flag bit separately

```
Retired   Equ   X'40'              Retired status flag bit
FullTime  Equ   X'20'              Full time worker status flag bit
PartTime  Equ   X'10'              Part time worker status flag bit
Exempt    Equ   X'08'              Exempt employee status flag bit
Hourly    Equ   X'04'              Hourly employee status flag bit
          - - -          etc.      - - -
          OI    Person_X,Retired   Person has retired [Better Method]
```


Avoiding Bit-Naming Problems

- Remember: bit names are *numbers*, not addresses!
- Example of a problem: Define a bit in each of two bytes:

```
Flag1    DS    X              |    Flag2    DS    X
Bit0     Equ   X'80'          |    Bit1     Equ   X'40'
```

- Normally we would write something like

```
         OI    Flag1,Bit0     |         OI    Flag2,Bit1
```

- But we could accidentally write (without assembler error!)

```
         OI    Flag2,Bit0     |         OI    Flag1,Bit1
```

- One way to associate specific bits with their "owning" bytes:

```
Fulltime Equ   *,X'20'      Assign a location and length attribute
PartTime Equ   *,X'10'      ... For each bit
         - - -
         DS    X            Now define the (unnamed) owning byte
```

- Reference a bit using its location *and* its length attribute; then each named bit is firmly attached to its owning byte

```
         TM    Fulltime,L'Fulltime     [Best!]
```


- You may see (or be tempted to write) self-modifying programs
  1. Skip some statements after they're executed once

```
NOP       NOP   SkipIt              Fall through first time here
          OI    NOP+1,X'F0'         Change NOP to unconditional branch
          - - -
SkipIt    DC    0H                  Continue execution here
```

  2. Alternating between branching or not

```
          - - -
          XI    Switch+1,X'F0'      Alternate branch masks at 'Switch'
Switch    BC    15,SomeWhereElse    Mask = 0, 15, 0, 15, ...
```

- This is a poor practice:
  1. Serious negative impact on performance
  2. The program can't be shared in memory
  3. You may not be debugging the program in the listing
- Advice: use a bit flag in a data area

- These are the Storage-Immediate instructions described in Section 23:

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

- Section 24 introduces key aspects of important SS-type instructions:
  - machine instruction and Assembler Language operand formats
  - operand formats with explicit and implicit length specifications
  - using Length Attribute References
  - Program vs. Encoded lengths, and why they′re different
  - three MOVE CHARACTERS instructions
  - logical AND, OR, and XOR instructions
  - logical comparison
  - the TR (Translate) instruction
  - Translate and Test instructions
  - the powerful and flexible Execute instructions

- Start with these typical SS-type instructions:

Mnem	Instruction	Mnem	Instruction
MVC	Move [Characters]	MVCIN	Move [Characters] Inverse
NC	AND [Characters]	OC	OR [Characters]
XC	XOR [Characters]	CLC	Compare Logical [Characters]
TR	Translate	TRT	Translate and Test
TRTR	Translate and Test Reverse		

- Each has this machine-instruction format:

opcode	**L**	B_1	D_1	B_2	D_2

- The basic assembler instruction statement format is:

  mnemonic D_1(N,B_1),D_2(B_2)

  - We'll see how **L** and **N** differ in Section 24.5
  - Only TRT and TRTR change general registers (only GR1, GR2)

- A machine instruction operand can have one of 3 formats:

  expr expr(expr) expr(expr,expr) or expr(,expr)

- For SS-type instructions,
  1. Operand 1 can have any of the formats
  2. Operand 2 can have only the first and second formats
- If *you* specify an *explicit* length **N**, valid operand forms are:

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)

- Examples:

```
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)
```


- Written as L′ followed by a symbol

```
LA    0,L′BB          C(GR) = Length Attribute of BB
```

1. Symbols defined in EQU statements with * or a self-defining term as operand have length attribute 1
   - If the EQU has a second operand, its value is the length attribute of the symbol

```
G         Equ   *             Length attribute of G = 1
H         Equ   *,20          Length attribute of H = 20
```

2. Literals have the length attribute of the first operand

```
LA    0,=X′123456,ABC,FEDCBA98′   c(GR0) = 3
```

3. The length attribute of a Location Counter Reference (*) is the length of the instruction in which it appears

```
MVC   BB(L′*),AA      Length attribute of MVC = 6
```


- If you don't specify length **N**, the assembler assigns an *implied length*

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)

- Implied lengths simplify specifying how many bytes are involved

```
MVC   AA,BB          Move L'AA bytes from BB to AA
```

- Summary of explicit/implied addresses and lengths

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

- **L** is one less than **N**, unless **N** is zero; then **L** is zero also
- Why use two notations (**N** and **L**) for lengths?
  1. You want to specify the true number of bytes involved, **N**
  2. The CPU needs to see a number one less:
     - The Effective Address+**L** is the address of the operand's rightmost byte
       - Some instructions operate from right to left
  3. The useful Execute instructions (Section 24.11) need **L**=0
- **Warning!** The *z/Architecture Principles of Operation* uses "**L**" for *both* **N** and **L**!
- Remember: **L** = **N**−1 unless **N**=0; then **L**=0 also

Mnem	Instruction	Mnem	Instruction
MVC	Move [Characters]	MVCIN	Move [Characters] Inverse

- These instructions move 1-256 bytes ($0 \leq$ **L** ≤ 255)
- Examples of MVC

```
MVC   AA(23),BB          Move 23 bytes from BB to AA
MVC   AA,BB              Move L'AA bytes from BB to AA

MVI   Line,C' '          Move a single blank to Line
MVC   Line+1(120),Line   Propagate blanks to fill 121 bytes

MVC   Str(40),Str+2      Shift 40 bytes left 2 positions
MVC   Str+40(2),=C' '    Replace last two bytes at Str by spaces
```

- Example of MVCIN: the second operand is moved in reverse order to the first; the second operand address is of the rightmost byte

```
          MVCIN RevData,Data+L'Data-1   Move reversed from Data to Revdata
          - - -
Data      DC    C'12345'                Source operand
RevData   DC    CL(L'Data)              Target operand; result = C'54321'
```


- These instructions perform a logical operation between corresponding bytes of the first and second operands, and set the CC:

Operation	CC setting
AND OR XOR	**0:** all result bits are zero **1:** result bits are not all zero

- AND: branch to Z if the word at W is zero:

```
NC    W,W          AND each byte to itself
JZ    Z            Branch if all bytes are zero
```

- OR: branch to Z if the word at W is zero:

```
OC    W,W          OR each byte to itself
JZ    Z            Branch if all bytes are zero
```

- XOR: set the at W to zero:

```
XC    W,W          XOR each byte with itself
```


- CLC compares two byte strings as unsigned 8-bit integers
  - Any inequality stops the comparison

CC	Indication
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

- Example: 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
```

  - Example: Compare the non-negative words at A and B, and branch to `AHigh`, `ALow`, or `ABEqual` accordingly

```
         CLC   A,B           Compare two non-negative integers
         JH    AHigh         Branch if c(A) > c(B)
         JL    ALow          Branch if c(A) < c(B)
         J     ABEqual       Branch if c(A) = c(B)
```


- TR replaces each first-operand byte with a second-operand byte, one byte at a time
  - The 8-bit binary value of a 1st-operand “argument” byte gives the offset to a 2nd-operand “function” byte
  - The function byte replaces the argument byte
  - The Condition Code is unaffected
- Example: replace all non-numeric characters at `Data` with blanks

```
          TR    Data,BlankTbl           Replace non-numerics with blanks
          - - -
BlankTbl  DC    240C' ',C'0123456789',6C' '  Translate table
Data      DC    C'A1G2p3?4+5W6/7'       Result = C' 1 2 3 4 5 6 7'
```


- TRT and TRTR *test* 1st-operand bytes using 2nd-operand byte values; the 1st operand is unchanged
  - The value of an “argument” byte is the offset to a “function” byte
  - If the function byte is zero, continue. Otherwise:
    1. Put the function byte in the rightmost byte of GR1
    2. Put the *address* of the argument byte in GR2, and stop scanning
    3. Set the Condition Code:

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.

- Example: scan the table at `Data` for numeric characters

```
          TRT   Data,NumTable      Scan Data for numeric characters
          - - -
NumTable  DC    240X'0',10X'1',6X'0'    Detect numeric characters
Data      DC    C'A1G2p3?4+5W6/7'  Data to be scanned for numerics
```


- EX and EXRL are often used with SS-type instructions

Mnem	Instruction	Mnem	Instruction
EX	Execute	EXRL	Execute Relative Long

1. Save the R_1 digit of the Execute instruction
2. Put the instruction at the Effective Address in the Instruction Register (IR) in place of the Execute instruction
   - The Instruction Address (IA) in the PSW remains unchanged
3. If the instruction in IR is an Execute, cause a program interruption
4. If the R_1 digit is nonzero, OR the rightmost digit of GR R_1 into the second byte of the IR
5. Execute the instruction in the IR

- Any CC settings are due to the executed instruction
- The R_1 digit is nonzero for almost all uses of Execute instructions

- Example 1: Move a message to `Line` whose address and length are in GR8 and GR9 respectively

```
          BCTR  9,0              Reduce length N in GR9 by 1 (L=N-1)
          EX    0,MoveMsg        Move the message text to Line
          - - -
MoveMsg   MVC   Line(*-*),0(8)   Move text at GR8 address to Line
```

- Example 2: The fullword at `Mask` contains an integer whose value lies between 0 and 15; use it as the mask digit of a BC instruction branching to `CondMet`

```
          L     1,Mask           Get mask value
          SLL   1,4              Position correctly for use as M1
          EX    1,BCInst         Execute the BC
NotMet    - - -                  Fall through if condition not met
          - - -
BCInst    BC    0,CondMet        BC with mask of 0
```

  - Note that if the branch condition is met, control will be taken from the EX instruction

- Instructions discussed in Section 24:

Function	Instruction	Data is Processed	CC Set?
Move	MVC MVCIN	Left to right Right to left	No
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

- Section 25 describes some *interruptible* instructions that may end before processing is complete
- The CPU supports resumption in two ways:

  A. Update registers to reflect current progress; reset the IA in the PSW to the address of the interrupted instruction

  - When processing resumes the instruction continues as if no interruption had occurred

  B. Update registers to reflect current progress; set CC=3 and terminate the instruction

  - An interruption may or may not occur at this point
  - When processing resumes, the *following* instruction tests for CC=3 and branches back to the terminated instruction to continue its task

- Section 25 instructions use both methods

Mnem	Instruction	Mnem	Instruction
MVCL	Move Long	CLCL	Compare Logical Long

MVCL R_1,R_2 and CLCL R_1,R_2

- Both instructions use Method “A” when interrupted, and two even-odd register pairs
  - The even-numbered register holds the operand address
  - The odd-numbered register holds the true operand length (0-$2^{24}-1$ bytes)
  - The operands may have different lengths
- The high-order byte of R_2+1 holds a *pad* byte
- All four registers may be updated by the instructions
- Both instructions set the CC
  - MVCL sets CC=3 and moves no data if destructuve overlap is possible:
    - part of the target field is used as source data after data has been moved into it

- Conceptually, MVCL works like this:
  1. As each byte is moved addresses are incremented, lengths decremented
  2. If both lengths=0 at the same time, set CC=0
  3. If the target length c(R_1+1) is 0 before the source length c(R_2+1), set CC=1
  4. If the source length c(R_2+1) is 0 before the target length c(R_1+1), use the pad character as source data until the target length is 0; set CC=2

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

- Example: Set 2400 bytes at **Field** to zeros

```
         LA    0,Field       c(R1) = Target address
         LHI   1,2400        c(R1+1) = Target length
         SR    3,3           c(R2+1) = Source length = 0; pad = X'00'
*        No source address is required if source length is zero
         MVCL  0,2           Move X'00' pad bytes to target Field
```


- Conceptually, CLCL works like this:
  1. Compare pairs of bytes; decrement addresses, increment lengths
  2. If both lengths=0 at the same time, set CC=0
  3. If an inequality is found, R_1 and R_2 contain the addresses of the unequal bytes; set CC=1 or CC=2
  4. If either length is 0, compare bytes from the longer operand to the pad byte

CC	Meaning
0	Operand 1 = Operand 2, or both lengths 0
1	First Operand low
2	First Operand high

- Example: Branch to **Cleared** if the 2400 bytes at **Field** are zeros

```
         LA    0,Field       c(R1) = Target address
         LHI   1,2400        c(R1+1) = Target length
         SR    3,3           c(R2+1) = Source length = 0; pad = X'00'
*        No source address is required if source length is zero
         CLCL  0,2           Compare target Field bytes to X'00'
         JE    Cleared       Branch if the Field was all zeros
```


Mnem	Instruction	Mnem	Instruction
MVCLE	Move Long Extended	CLCLE	Compare Logical Long Extended

- MVCLE and CLCLE generalize MVCL and CLCL. Their form:

opcode	R_1	R_3	B_2	DL_2	DH_2	opcode

- The R_1 and R_3 operands are like R_1 and R_2 for MVCL and CLCL; addresses and lengths depend on addressing mode
  - The low-order byte of operand 2 is the pad character (*not* an address!)
  - The odd-numbered registers hold 32- or 64-bit lengths (depending on addressing mode) vs. 24-bit lengths for MVCL/CLCL
- Assembler Language syntax

```
mnemonic  R1,R3,D2(B2)     Target,source,pad_character
```

as in

```
MVCLE 2,8,C' '(0)          Pad character = C' '
CLCLE 4,14,X'40'(0)        Pad craracter = X'40'
```


- Conceptually, MVCLE works like this:
  1. As each byte is moved, increment addresses, decrement lengths
  2. If both lengths=0 at the same time, set CC=0
  3. If the target length c(R_1+1) is 0 before the source length c(R_3+1), set CC=1
  4. If the source length c(R_3+1) is 0 before the target length c(R_1+1), use the pad character as source data until the target length is 0; set CC=2

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	CPU wants to rest; branch back to the MVCLE

- Example: set 2400 bytes at **Field** to zeros

```
         LA    0,Field       c(R1) = Target address
         LHI   1,2400        c(R1+1) = Target length
         SR    3,3           c(R3+1) = Source length = 0
         SR    5,5           c(R5) = Pad byte = X'00'
*        No source address is required if source length is zero
         MVCLE 0,2,0(5)      Move pad bytes to target Field
```


- Conceptually, CLCLE works like this:
  1. Compare pairs of bytes; decrement addresses, increment lengths
  2. If both lengths=0 at the same time, set CC=0
  3. At inequality, R_1 and R_3 address the unequal bytes; set CC=1 or CC=2
  4. If a length is 0, compare bytes from the longer operand to the pad byte

CC	Meaning
0	Operand 1 = Operand 2, or both 0 length
1	First operand low
2	First operand high
3	No inequality found thus far; operands are not exhausted

- Example: branch to **Cleared** if the 2400 bytes at **Field** are zeros

```
         LA    0,Field       c(R1) = Target address
         LHI   1,2400        c(R1+1) = Target length
         SR    3,3           c(R2+1) = Source length = 0
         SR    5,5           Set pad character to X'00'
*        No source address is required if source length is zero
         CLCLE 0,2,0(5)      Compare target Field bytes to X'00'
         JE    Cleared       Branch if the Field was all zeros
```


- Character strings in C/C++ ("C-strings") end with a null (X'00') byte
  - We sometimes use a bold-italic "***n***" to represent a null byte

```
CString  DC    C'A C-string.',X'0'      Generates 'A C-string.n'
```

- These four instructions simplify working with C-strings:

Mnem	Instruction	Mnem	Instruction
MVST	Move String	CLST	Compare Logical String
SRST	Search String	TRE	Translate Extended

- Each has Assembler Language syntax

```
mnemonic  R1,R2
```

- Each requires a special "end" or "test" character in the rightmost byte of GR0; it can be *any* character
  - TRE also requires a length operand
- Each uses Method B to handle interruptions
- They have many uses beyond C-strings

- SRST searches a string of bytes for a match of the test character
  1. GR0 is zeroed; the test character is placed in its rightmost byte
  2. The start of the string is placed in R_2
  3. One byte past the end of the string is placed in R_1 (to limit the search)
- Condition Code settings:

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

  - Usually, SRST is faster searching for single characters than a CLI loop or TRT
- Example: Search a byte string at **Expr** for a left parenthesis

```
LHI   0,C'('          Test character
LA    4,Expr          Start of string
LA    8,Expr+L'Expr   One byte past end of string
SRST  8,4             Search for '(' at Expr
```


- MVST moves a C-string, including the test character
  1. GR0 is zeroed; the test character is placed in its rightmost byte
  2. The target-string address is placed in R_1
  3. The source-string address is placed in R_2
- The Condition Code settings are:

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

- Example: Move a C-string from **Here** to **There**

```
XR    0,0          Test character is a null byte
LA    7,There      Target address
LA    1,Here       Source address
MVST  7,1          Move from Here to There
```

- For very long strings with known lengths, MVCL or MVCLE may be faster

- CLST compares two strings terminated with the *same* stop character
- Comparison stops when an inequality is detected, or the end of an operand is reached
- A shorter operand is always considered “low” compared to the longer
- Condition Code settings:

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

- Example: Compare the C-strings at **Before** and **After**

```
SR    0,0          Compare null-terminated C-strings
LA    3,Before     Address of first operand string
LA    6,After      Address of second operand string
CLST  3,6          Compare the two strings
```


- TRE is similar to TR, but more flexible:
  1. Translated string address is the R_1 operand, translate table address is the R_2 operand
     - The string length is in (odd register) R_1+1
  2. GR0 is zeroed; the test character is placed in its rightmost byte
  3. TRE stops when (a) all bytes are translated, or (b) a source byte (which is *not* translated) matches the stop character
- Condition Code settings:

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

- CUSE searches for common substrings of a specified length
- The matching substrings must be at the *same* offset in both strings

  'ABCDEFG' and 'QRSDEFT' – matching strings at offset 3: lengths 1, 2, or 3

  'ABC' and 'BCD' – if pad='*', matching substring at offset 3: length 2**

  1. Operand addresses are in even-numbered registers R_1 and R_2; Operand lengths are in corresponding odd-numbered registers R_1+1 and R_2+1
  2. The rightmost bytes of GR0 and GR1 contain the desired substring length and the padding byte, respectively
- Condition Code settings:

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 incomplete, last compared bytes unequal; R_1, R_2, lengths are updated

- Instructions discussed in this section are summarized in this table:

Function	Length control	End-char control
Move	MVCL MVCLE	MVST
Compare	CLCL CLCLE CUSE	CLST
Search		SRST
Translate	TRE	

- Use care with null-terminated C-strings: if the terminating null byte is omitted, programs scanning or moving such strings may “process” far more data than intended

Section 26 investigates forms of character data other than “Assembler Language EBCDIC”

- 6-bit “Binary Coded Decimal” (BCD)
- Some of the many alternative EBCDIC representations
- “American Standard Code for Information Interchange” (ASCII)
- Double-byte EBCDIC and its Assembler Language representation
- Unicode, a universal encoding
  - Instructions tailored to Unicode data
  - Transformation formats
- Byte reversal instructions and workstation data

- Computers that process character data must know how it's represented
  - A group of bits can represent a number or a character
  - A defined number-character correspondence is an *encoding*
- Binary Coded Decimal (BCD) was used on early IBM machines
- 8-bit "Assembler Language" EBCDIC was used in many earlier examples
  - Many other EBCDIC encodings have been defined
- The spread of digital technology has led to other encodings
  - ASCII, "American Standard Code for Information Interchange"
  - Double-Byte EBCDIC for ideographic scripts
  - Unicode, and attempt to encode all known characters

- The worldwide use of IBM “mainframes” required added character support
  - Each set of 256 encodings is called a *Code Page*
  - “Assembler Language EBCDIC” is Code Page 037
- Other EBCDIC code pages support national characters like á, ä, ç, Ø, Ω
  - Many characters have different encodings in different code pages
- The “Syntactic Character Set” has the same encodings across EBCDIC code pages:
  - blank, decimal digits, lower and upper case alphabetics, and

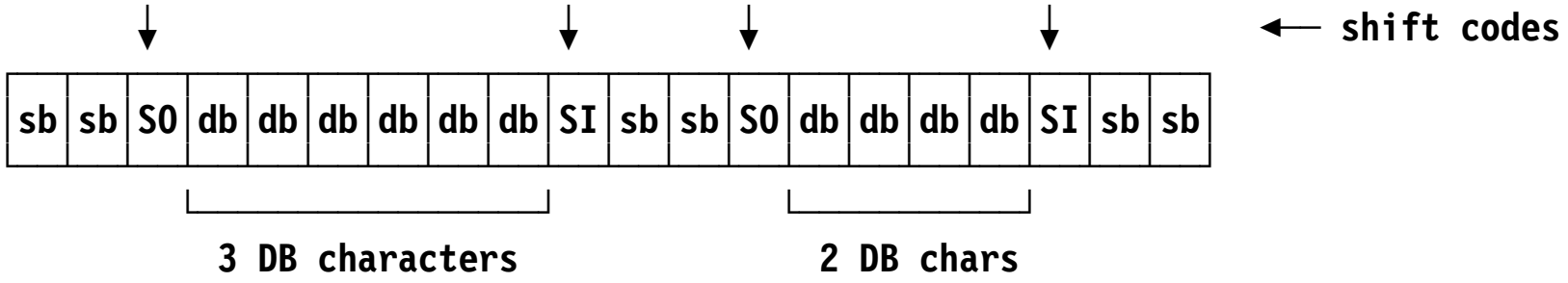
    + < = > % & * ″ ′ ( ) , _ – . / : ; ?

  - It does *not* include # @ $ (allowed in Assembler Language symbols)
- All modern EBCDIC code pages support the “euro” character ∈

- Widely used on non-System z computers
  - Basic encoding is 7 bits wide (X'00'-X'7F')
    - First 32 positions (X'00'-X'1F') are reserved for control codes
- Decimal digits are X'30'-X'39'
- Upper-case letters X'41'-X'5A'; lower-case X'61'-X'7A'
- For ASCII character constants, use subtype A:

```
DC     CA'ASCII'          Generates X'4153434949'
```


- Double-Byte characters have special coding rules
  - Groups of DBCS byte pairs always enclosed in Shift-Out, Shift-In bytes
    - Shift-Out (X'0E', "SO") shifts *out* of single-byte mode to double-byte mode
    - Shift-In (X'0F', "SI") shifts *in* to single-byte mode from double-byte mode
- Example: mixing single-byte EBCDIC ("sb") and DBCS ("db") characters



- Assembling DBCS data requires the DBCS option
  - G-type constants and self-defining terms may be needed
- Used most often for representing Japanese characters

- All Unicode characters can have any of 3 formats:
  - UTF-8: an encoding is 1-4 bytes long
  - UTF-16: most characters are 2 bytes long; some are 2 2-byte pairs
  - UTF-32: all characters are 4 bytes long
- Instructions can convert any encoding to any other
- UTF-16 is most widely used; notation is U+nnnn where "nnnn" is 4 hex digits
  - The encoding of U+nnnn is X'nnnn'
  - ASCII encodings have values from U+0000 to U+00FF
    - Encodings U+0000 - U+FFFF are known as the "Basic Multilingual Plane"
- Unicode constants are written with type extension U, and generate UTF-16 characters:

```
DC    CU'Unicode '  Generates X'0055006E00690063 006F006400650020'
```


- We will discuss the instructions in three groups:
  1. String search, compare, and move instructions

Mnem	Instruction	Mnem	Instruction
SRSTU	Search String Unicode	CLCLU	Compare Logical Long Unicode
MVCLU	Move Long Unicode		

  2. Translation instructions

Mnem	Instruction	Mnem	Instruction
TROO	Translate One to One	TROT	Translate One to Two
TRTO	Translate Two to One	TRTT	Translate Two to Two

  3. Format conversion instructions

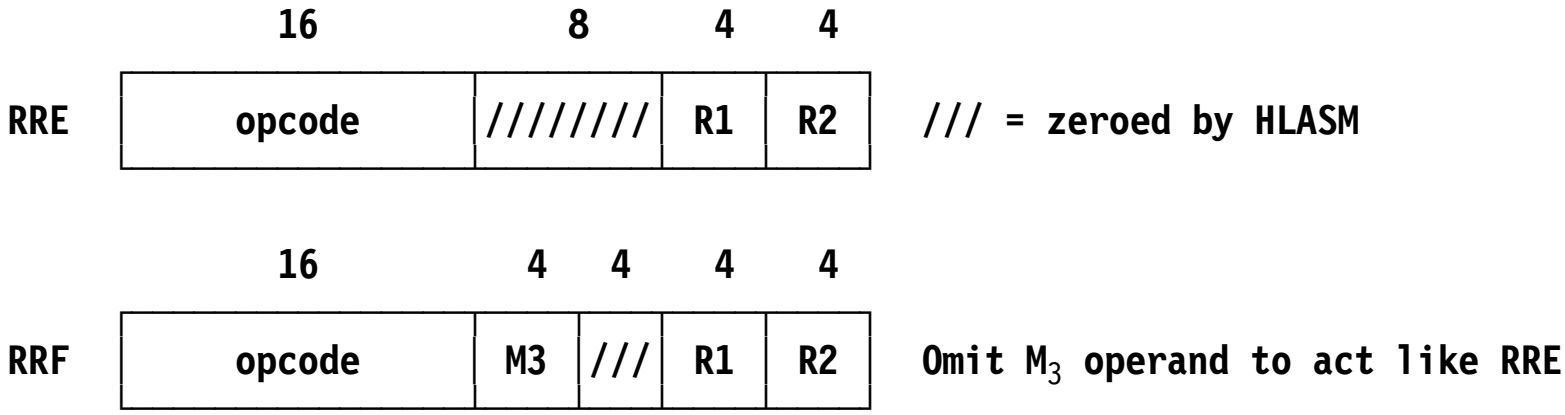
Mnem	Instruction	Mnem	Instruction
CU12, CUTFU	Convert UTF-8 to UTF-16	CU14	Convert UTF-8 to UTF-32
CU21, CUUTF	Convert UTF-16 to UTF-8	CU24	Convert UTF-16 to UTF-32
CU41	Convert UTF-32 to UTF-8	CU42	Convert UTF-32 to UTF-16

- These instructions are equivalent to the similar single-byte instructions (SRST, MVCLE, CLCLE)

  But they handle *pairs* of bytes (which need not be Unicode characters, nor halfword aligned)

- **SRSTU**: scan a string addressed by R_2 for a byte pair matching the rightmost 2 bytes of GR0; R_1 has the address of the first byte after the string
  - R_2 incremented by 2 for each comparison
- **MVCLU**: moves pairs of bytes from area addressed by R_3 to area addressed by R_1; lengths in R_1+1, R_3+1; "padding pair" is low-order 16 bits of $D_2(B_2)$ Effective Address
  - Each pair moved increments addresses by 2, decrements lengths by 2
  - Special padding rules if any length is odd
- **CLCLU**: registers and "padding pair" assigned like MVCLU's
  - Each pair compared increments addresses by 2, decrements lengths by 2
  - Lengths must be even

- Problem for CPU architects: how to enhance an existing instruction without creating incompatibilities?
  1. Create a new instruction (but there are quite a few already...)
  2. Use previously empty fields that were set to zero by HLASM
     - Make new operands optional: if omitted, same behavior as before
- Example: RRE- and RRF-type instructions



- Assembler instruction format:

mnemonic R_1,R_2[,M_3] [ ] indicates optional operand

- Sometimes need to translate to, from, or among Unicode encodings
- Four instructions:

  TROO: Translate One to One (like TR but much more flexible)

  TROT: Convert single-byte data to double-byte (e.g. EBCDIC or ASCII to Unicode)

  TRTO: Convert couble-byte data to single-byte (e.g. Unicode to EBCDIC or ASCII)

  TRTT: Convert among double-byte data formats to Unicode)

- All four instructions have an optional M_3 operand

  TRxx R_1,R_2[,M_3]

- Uses aren't limited to character data!

- Unicode characters have 8-, 16-, and 32-bit formats called UTF-8, UTF-16, and UTF-32
  - UTF-8 format is complex; used only for network transmission
- Six instructions for conversion among formats

Mnem	Instruction	Mnem	Instruction
CU12, CUTFU	Convert UTF-8 to UTF-16	CU14	Convert UTF-8 to UTF-32
CU21, CUUTF	Convert UTF-16 to UTF-8	CU24	Convert UTF-16 to UTF-32
CU41	Convert UTF-32 to UTF-8	CU42	Convert UTF-32 to UTF-16

- Operand formats:

```
CUxx  R1,R2[,M3]  For CU12, CU14, CU21, CU24
CUxx  R1,R2       For CU41, CU42
```

  - Initial implementations (CUTFU, CUUTF) did no “well-formedness” tests
  - If M_3=1, invalid operand data sets CC=2

- Each instruction uses an optional operand to create six “additional” instructions with a single opcode
- Operand format:

```
Mnemonic  R1,R2[,M3]   M3 bits are B'AFLO'
```

- M_3 mask bits:

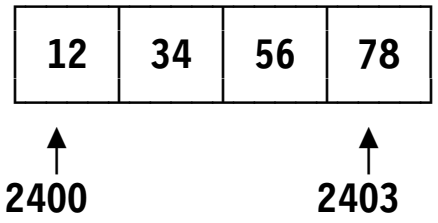
  A 0: Argument characters are 1 byte
  1: Argument characters are 2 bytes

  F 0: Function codes are 1 byte
  1: Function codes are 2 bytes

  L 0: Full range of argument and function codes allowed
  1: Argument > 255 means function code assumed to be zero

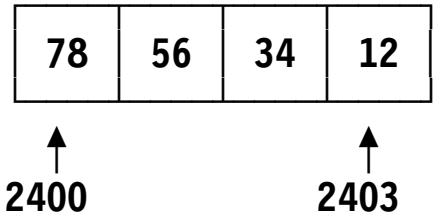
- Mask bits provide greate flexibility
  - But not all 9 A-F-L combinations are meaningful...

- Suppose a 32-bit integer X'12345678' starts at address X'2400'.



IBM System z ("Big-Endian")

- On some processors (e.g. Intel) it's stored like this:



Some workstations ("Little-Endian")

Mnem	Instruction	Mnem	Instruction
LRV	Load Reversed (32)	LRVR	Load Register Reversed (32)
LRVG	Load Reversed (64)	LRVGR	Load Register Reversed (64)
LRVH	Load Halfword Reversed (16)	STRVH	Store Halfword Reversed (16)
STRV	Store Reversed (32)	STRVG	Store Reversed (64)

Chapter VIII: Zoned/Packed Decimal Data and Operations 1

This chapter explores the the zoned and packed decimal representations and operations on them

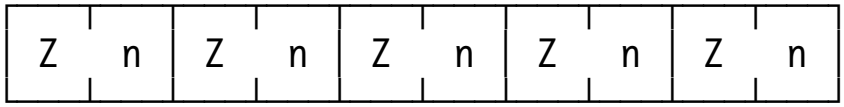
- Section 27 describes the zoned and packed representations in detail, and instructions to convert between them
- Section 28 investigates the operations of packed decimal comparison, addition and subtraction, multiplication, and division to prepare for the instructions in Section 29
- Section 29 discusses the instructions that test, move, compare, shift, and do arithmetic operations on packed decimal operands
  - Scaled arithmetic for values with fractional parts is discussed in Section 29.10
- Section 30 examines techniques and instructions for converting among binary, packed decimal, and character formats

- First, we describe the zoned decimal representation
  - It is quite close to “normal” EBCDIC characters
- Next we examine the packed decimal representation
- Then we discuss instructions for converting data between zoned and packed decimal formats

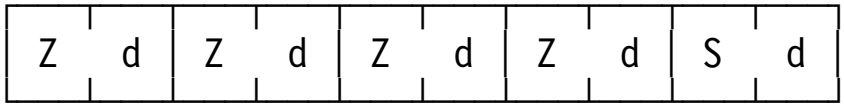
Mnem	Instruction	Mnem	Instruction
MVN	Move Numerics	MVZ	Move Zones
PACK	Pack	UNPK	Unpack
PKA	Pack ASCII	UNPKA	Unpack ASCII
PKU	Pack Unicode	UNPKU	Unpack Unicode

  - All these instructions have SS-1 or SS-2 format

- Notation used for bytes of any type:
  left hex digit of a byte is the “zone” digit (Z);
  the right hex digit is the “numeric” digit (n)



- Two special move instructions, very much like MVC:
  1. MVN: moves only the numeric digits; source and target zone digits are untouched
  2. MVZ: moves only the zone digits; source and target numeric digits are untouched
- Internal representation of zoned decimal digits is



Z=zone digit, d=decimal digit, S=sign code

- Sign codes: (+) A, C, E, F; (−) B, D. (Preferred codes are C, D)

- Defined using constant type **Z**

```
ZCon1    DC    Z'1470369258'  Generates X'F1F4F7F0F3F6F9F2F5C8'
ZCon2    DC    Z'-1'          Generates X'D1'
ZCon3    DC    Z'5,+6,-749'   Generates X'C5C6F7F4D7'
ZCon4    DC    ZL5'29'        Generates X'F0F0F0F2C9'
```

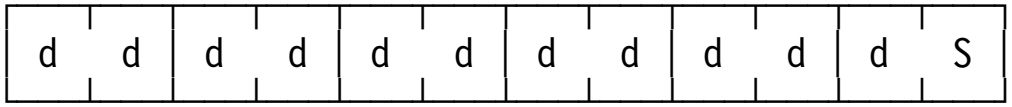
- Only a length modifier is valid (no integer, scale, exponent)
- Decimal points in nominal values are ignored

```
ZCon5    DC    Z'1234.5'      Generates X'F1F2F730F4C5'
ZCon6    DC    Z'1.2345'      Generates X'F1F2F730F4C5'
```

  - The Assembler assigns Integer and Scale attributes
    - But the meaning of a decimal point is up to you!

Often used for business, financial calculations

- "Packed" because there are 2 binary-coded decimal digits per byte
  - The rightmost byte has a decimal digit (left half) and a sign code (right half)



  - An N-byte packed decimal field holds 2N-1 digits
  - Sign code **S** is the same as for zoned decimal
  - Examples (Note: no zones, just digits and a sign code)

Value	Representation
+12345	X'12345C'
-0012345	X'0012345D'
+3	X'3C'
-09990	X'09990D'
39	X'039C'

- Defined with constant type **P**

```
PCon1    DC    P'12345'      Generates X'12345C'
PCon2    DC    P'-27,+62'    Generates X'047D062C'
PCon3    DC    PL4'999'      Generates X'0000999C'  (Padded on left)
PCon4    DC    PL2'12345'    Generates X'345C'      (Truncated on left)
```

- Only a Length modifier is allowed
- Decimal points in nominal values are ignored in generated constants

```
PCon5    DC    P'1234.5'     Generates X'12345C'
PCon6    DC    P'1.2345'     Generates X'12345C'
```

  - HLASM assigns Integer and Scale attributes:

```
PCon5:  Integer attribute = 4, Scale attribute = 1
PCon6:  Integer attribute = 1, Scale attribute = 4
```


- Use the **PACK** and **UNPK** instructions
- Both have Assembler Language syntax

mnemonic $D_1(N_1,B_1),D_2(N_2,B_2)$

- Machine instruction format:

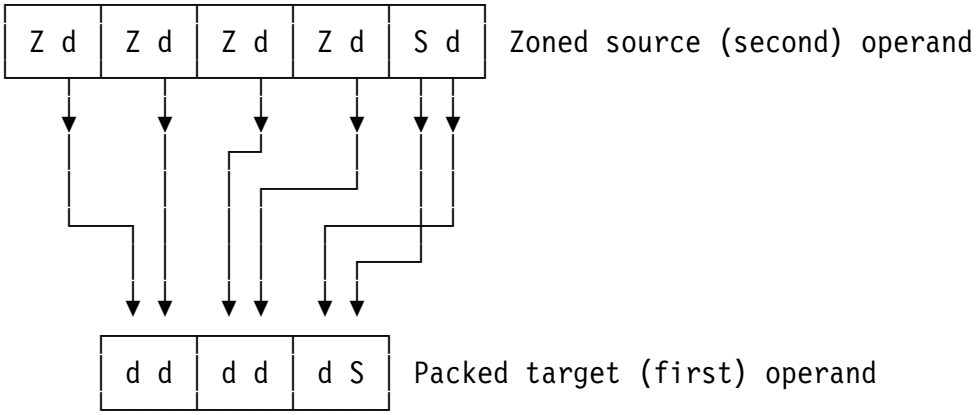
opcode	L_1	L_2	B_1	D_1	B_2	D_2

- Encoded Lengths L are one less than Program Lengths N
- Each operand can take one of four forms:

	Implied Length	**Explicit Length**
Implied Address	S_1	$S_1(N_1)$
Explicit Address	$D_1(,B_1)$	$D_1(N_1,B_1)$

- **PACK** converts from zoned to packed, working from right to left

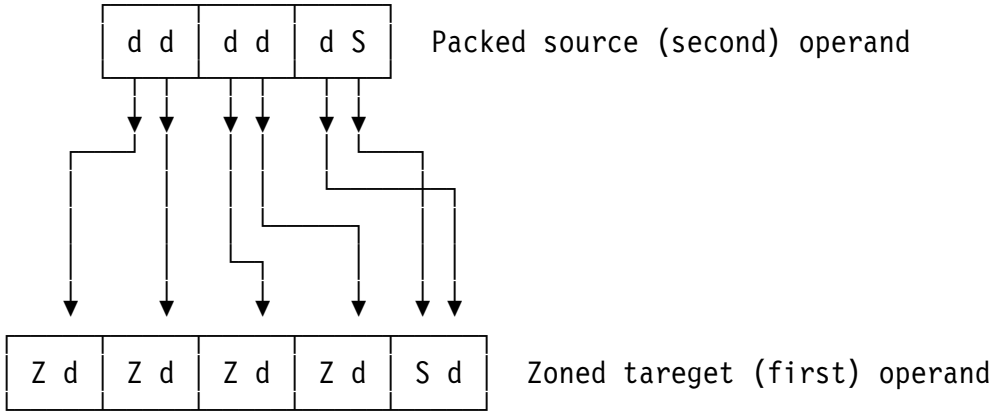
```
PACK  Target,Source
```



- Padding/truncation rules for first (target) operand:
  - Source too short: pad target on left with zero digits
  - Source too long: stop packing when target field is full
- Condition Code is unchanged
  - Overlapping operands produce predictable results!

- **UNPK** converts from packed to zoned formats, working from right to left

```
UNPK  Target,Source
```



- Padding/truncation rules for first (target) operand:
  - Source too short: pad target on left with zoned zeros (X′ F0′)
  - Source too long: stop unpacking when target field is full
- Condition Code is unchanged
  - Overlapping operands generate predictable results!


```
Pack ASCII:        PKA   PDTarget,ASCII_Source(N)
Pack Unicode:      PKU   PDTarget,Unicode_Source(N)

Unpack ASCII:      UNPKA ASCII_Target(N),PDSource
Unpack Unicode:    UNPKU Unicode_Target(N),PDSource
```

- The packed decimal operand is *always*16 bytes long (31 digits)
- ASCII decimal digits: X'30-39' ; Unicode decimal digits: X'0030-0039'
- Instruction format for all four instructions:

  - **L** is the Encoded Length of the character operand

    Must be odd for Unicode (even number of bytes)

- For packing (PKA, PKU): L is length-1 of *second* (character) operand
  For unpacking (UNPKA, UNPKU): L is length-1 of *first* (character) operand

- It often helps to display data in hex
- Use UNPK and TR in these steps (we'll assume 4-byte data):

1. Move source data to right half of a work area:

```
         MVC   WorkArea+4(4),SourceData
         _ _ _
WorkArea DS    CL8,X          ← The extra byte is important!
```

2. Unpack **one** extra byte (at the right end):

```
         UNPK  WorkArea(9),WorkArea+4(5)   Extra byte is swapped
```

3. Translate the "spread hex" to EBCDIC characters

```
         TR    WorkArea,=C'0123456789ABCDEF'-C'0'
```

4. 8 bytes at **WorkArea** are ready for display or print

- Usually gives expected results
  - Operand size limitations for some operations
- Notation corresponds to internal representations

Packed: | 12 | 34 | 56 | 7D | **written 1234567–**

Zoned: | F1 | F2 | F3 | C4 | **written 1234+**

- Zoned values *must* be converted to packed for arithmetic

- Results of packed decimal operations replace the first operand
  - Division fits quotient and remainder in first-operand field
- Preferred signs (X'C', X'D') always given to results
- Overflow: set Condition Code to 3
  - If Program Mask bit is 1, cause Decimal Overflow interruption with Interruption Code X'000A'
- Invalid sign or numeric digits cause a Data Exception interruption with Interruption Code X'0007'
- Remember: packed decimal operands are treated as ***integers*** by System z

- Shorter operands are extended internally with high-order zeros
  - The result's significant digits must fit in the first operand field
  - If it won't, decimal overflow occurs; only low-order digits are kept
- Non-overflowed zero results always have a + sign

```
   003+          500−
 + 003−        + 500−
   000+          000−
(no overflow)  (overflow)
```

- Condition Code settings:

CC	Indication
0	Result is zero
1	Result is less than 0
2	Result is greater than 0
3	Decimal overflow

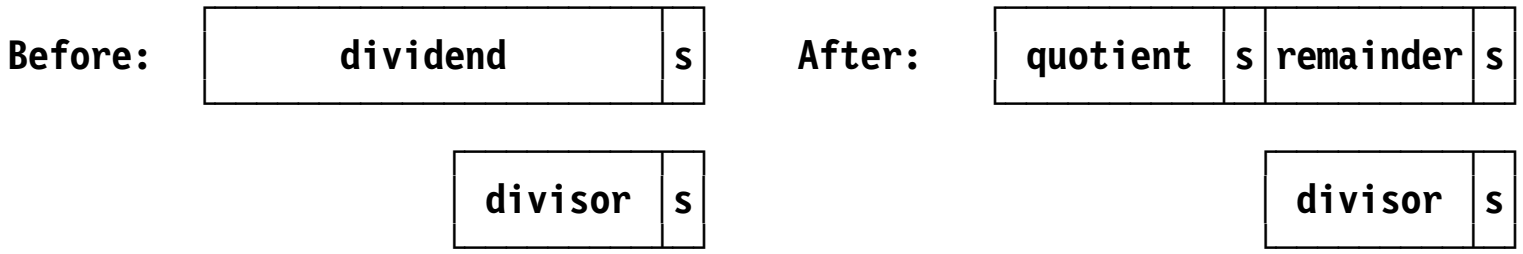
- Advice: avoid operand overlap

- Performs an internal subtraction
  - Operands extended with high-order zeros as needed
- 0+ treated as equivalent to 0-
- CC settings:

CC	Meaning
0	Operand 1 = Operand 2
1	Operand 1 < Operand 2
2	Operand 1 > Operand 2

- The product of N1-digit and N2-digit numbers is at most N1+N2 digits long
- The first operand must have at least as many high-order *bytes* of zeros as the number of bytes in the second operand
  - So operand 1 *must* be longer than operand 2
  - Operand 2 must be ≤ 8 bytes (15 digits) long
- Signs are determined by the rules of algebra
- The Condition Code is unchanged
- Warning: packed decimal products depend on the order of the operands

Decimal Division



- The remainder has the same bye length as the divisor
  - It always has the same sign as the dividend
- Quotient sign is determined by the rules of algebra
- Divisor length must be (a) ≤ 8 bytes, (b) < dividend length
- Division by zero or quotient too large causes a Decimal Divide exception with Interruption Code X'000B'
  - First operand is unchanged
  - Condition Code is unchanged; Decimal Overflow cannot occur

- The packed decimal instructions are

Mnem	Instruction	Mnem	Instruction
AP	Add Decimal	SP	Subtract Decimal
MP	Multiply Decimal	DP	Divide Decimal
CP	Compare Decimal	ZAP	Zero and Add Decimal
SRP	Shift and Round Decimal	MVO	Move with Offset
TP	Test Decimal		

- All but TP have 2-length SS-type format:

- All instructions process operands from *right* to *left*

- TP tests the validity of its operand. Assembler Language syntax:

  TP D_1(N,B_1)

- Its machine instruction format differs from the other instructions:

- Valid Assembler Language instruction operand formats:

	Explicit Length	Implied Length
Explicit Address	D_1(N,B_1)	D_1(,B_1)
Implied Address	S(N)	S

- Condition Code settings:

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.

- ZAP effectively (but not actually!)
  1. Sets the first operand to 0+
  2. Adds the second operand
- It can therefore generate
  - A data exception for an invalid second operand
  - A decimal overflow exception if the first operand is too short
- Assembler Language syntax

ZAP $D_1(N_1,B_1),D_2(N_2,B_2)$ or Target(N_1),Source(N_2)

- Condition Code settings

CC	Indication
0	Result is zero.
1	Result is less than zero.
2	Result is greater than zero.
3	Decimal overflow.

- The result replaces the first operand
- Assembler Language syntax:

```
AP    D1(N1,B1),D2(N2,B2)        (Same for SP)
```

- Condition Code settings:

CC	Indication
0	Result is zero.
1	Result is less than zero.
2	Result is greater than zero.
3	Decimal overflow.

- Overflow is possible if the first operand is too short

```
         AP    P123,P9      Result at P123 = 132+. CC=2
         AP    P9,P234      Result at P9   = 3+,   CC=3 (overlow)
P123     DC    P'+123'
P234     DC    P'+234'
P9       DC    P'+9'
```


- CP compares two packed decimal operands
  - Internal subtractions do not cause overflow
- Condition Code settings:

CC	Indication
0	Operands are equal.
1	First operand is low.
2	First operand is high.

- Examples:

```
CP    =P'+5',=P'+3'      CC=2
CP    =P'+3',=P'+5'      CC=1
CP    =P'+0',=P'-0'      CC=0
```


- Assembler Language syntax:

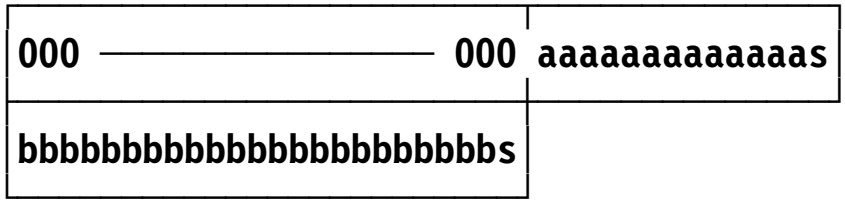
  MP $D_1(N_1,B_1),D_2(N_2,B_2)$

- Operand length restrictions:

  $2 \le N_1 \le 16$, $1 \le N_2 \le 8$, $N_1 > N_2$

  $1 \le L_1 \le 15$, $0 \le L_2 \le 7$, $L_1 > L_2$

- Important additional restriction:
  - There must be as many *bytes* of high-order zeros in the multiplicand (first operand) as the length of the multiplier (second operand); a specification exception otherwise



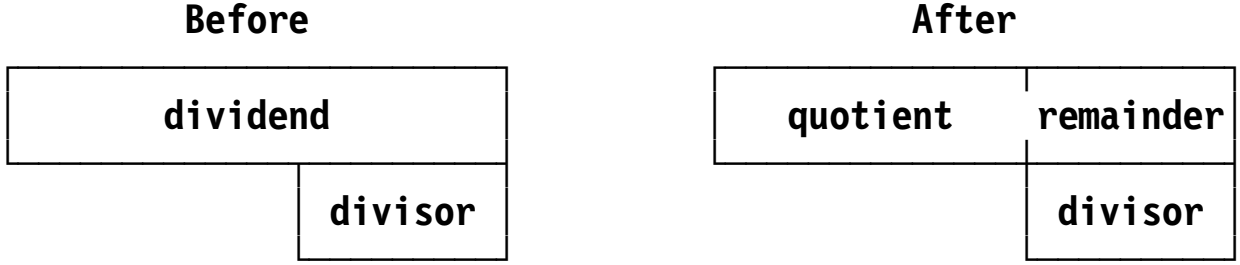
First operand (multiplicand)

Second operand (multiplier)

- Assembler Language syntax:

  DP $D_1(N_1,B_1),D_2(N_2,B_2)$

- Like binary division, the quotient and remainder replace the dividend (but in the opposite order)



  - Remainder and divisor have the same length
  - There must be at least one high-order zero in the dividend

- Operand lengths must obey the same restrictions as for MP:

  $2 \le N_1 \le 16, \quad 1 \le N_2 \le 8, \quad N_1 > N_2$

  $1 \le L_1 \le 15, \quad 0 \le L_2 \le 7, \quad L_1 > L_2$

- Example:

```
         ZAP   Dvnd,=P'162843'    Initialize dividend
         DP    Dvnd,=P'762'      Divide by 762+
         - - -
Dvnd     DS    PL4               Result: X'213C537C'
```


- SRP multiplies and divides by a power of 10, with optional quotient rounding
- Assembler Language syntax:

```
SRP   D1(N1,B1),D2(B2),I3
```

- Machine instruction format:

F0	L_1	I_3	B_1	D_1	B_2	D_2

- Shift amount *and* direction determined by low-order 6 bits of second-operand Effective Address:
  - B′100000′ = −32 ≤ shift count ≤ +31 = B′011111′
- Examples:

```
SRP   X,3,0          Multiply operand at X by 1000
SRP   X,64-3,5       Divide operand at X by 1000, round last digit
```

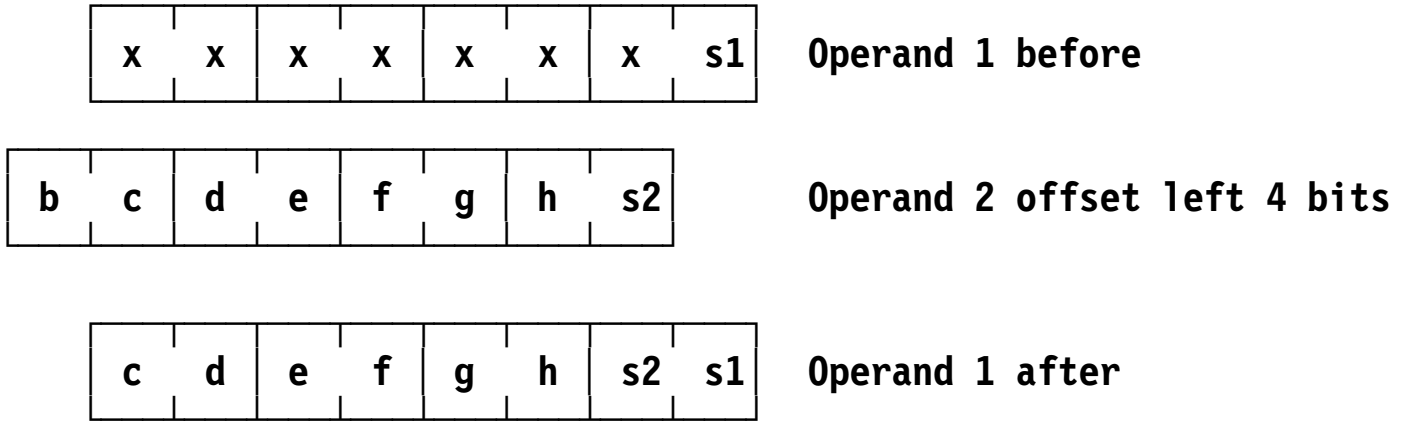
- Possible overflow on left shifts
- Rounded results are slightly biased

Move With Offset (MVO) Instruction 26

- MVO moves the second operand to the first, but offset to the left by 4 bits
- Assembler Language syntax:

```
MVO   D1(N1,B1),D2(N2,&B2)
```

- Example of two 4-byte operands with signs s1, s2:



- If $N_2 \geq N_1$, high-order digits are lost
- If $N_2 < N_1$, high-order digits positions are filled with zeros

- Before SRP was available, MVO was used for pscked decimal shifting
- Four different instruction sequences were required:
  1. Shift right an odd number of digits
  2. Shift left an odd number of digits
  3. Shift left an even number of digits
  4. Shift right an even number of digits
- These are rarely used today, but are instructive

- Scaled arithmetic uses values having decimal points not always following the rightmost digit
  - Most packed decimal arithmetic uses scaled operands
- Precision: number of digits in a value
  - *Not* the same thing as accuracy!
- The number of fraction digits is the *scale* of the value:

```
DC    P'123.4'      Precision = 4, Scale = 1 (Integer attribute = 3)
DC    P'5.678'      Precision = 4, Scale = 3 (Integer attribute = 1)
```

- If **I** = number of integer digits, and **F** = number of fraction digits, then Precision **P=I+F**, and Value = **I.F**
- Given operands 1 and 2 ($I_1.F_1$ and $I_2.F_2$), then:
  - Sum or Difference **I.F**: **I**=Max(I_1,I_2), **F**=Max(F_1,F_2)
  - Product **I.F**: **I**=I_1+I_2, **F**=F_1+F_2
  - Quotient **I.F**: **I**=I_1+F_2; for an N-digit result, **F**=N − **I**
- You must keep these in mind doing scaled packed decimal arithmetic

Section 30: Converting and Formatting Packed Decimal Data29

- These instructions help convert between binary and packed decimal, and from packed decimal to character
  - CVB/CVBY and CVD/CVDY differ only in their displacement's length and sign

Mnem	Instruction	Mnem	Instruction
CVB	Convert to Binary (32)	CVD	Convert to Decimal (32)
CVBY	Convert to Binary (32)	CVDY	Convert to Decimal (32)
CVBG	Convert to Binary (64)	CVDG	Convert to Decimal (64)
ED	Edit	EDMK	Edit and Mark

- Binary data is usually converted first to packed decimal
- ED and EDMK are powerful "programmable" instructions that convert packed to character
  - Their behavior is controlled by an *Edit Pattern* that you provide

- Convert 2's complement binary data in registers to packed decimal
- Example: suppose c(GR7) = X'00000087' (+135 in decimal)
  - CVD and CVDY convert 32-bit integers to 8 bytes of 15 packed decimal digits

```
          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'
```

- Example: suppose c(GG8) = X'4000000000000000' = 2^{62}
  - CVDG converts 64-bit integers to 16 bytes of 31 packed decimal digits.

```
          CVDG  8,WrkArea2    Convert to 16–byte packed decimal
          – – –
WrkArea2  DS    2D            Result=X'00000000 00004611 68601842 7387904C'
```

- The operands need not be aligned on any specific boundary
  - But doubleword alignment can improve performance

- Convert 8- or 16-byte packed decimal data to 2's complement binary in a register

```
          CVB   0,PackNum     Result in GR0 = X'FFFFFF79' = -135
          CVBG  9,PackNum2    Result in GG9 = X'FFFF8FB779F22087'
          - - -
PackNum   DC    0D,PL8'-135'
PackNum2  DC    0D,PL16'-123456789012345'
```

- Two interruptions are possible:
  1. Invalid decimal operands can cause a decimal data exception; the Interruption Code is set to 7
  2. If the packed decimal operands have values too large for a register, a fixed-point divide exception may occur; the Interruption Code is set to 9

```
          CVB   0,TooBig      Causes divide exception
          - - -
TooBig    DC    0D,PL8'123456789012345'      Exceeds 2**31 (somewhat)
```

- The operands need not be aligned on any specific boundary
  - Doubleword alignment can improve performance

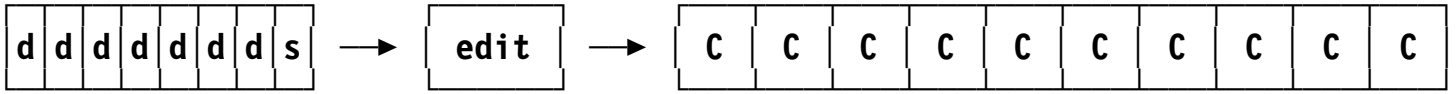
- Assembler Language syntax for ED and EDMK:

  mnem D_1(N,B_1),D_2(B_2) or Pattern(N),PackData

- Machine instruction format:

opcode	L	B_1	D_1	B_2	D_2

- The basic operation of the instructions:



Packed decimal data **ED, EDMK** **EBCDIC characters replacing pattern**

- Under control of the pattern (first operand), the instruction maps the signed or unsigned packed decimal data into EBCDIC characters
  - The editing process scans the pattern *once*, from left to right

- Editing actions depend on which pattern character (PC) is being processed, and
  - What happened previously, as determined by CPU's *Significance Indicator* (SI)
- There are five types of pattern characters (PCs):
  1. Fill Character (FC), may have any value; the first byte of the pattern
  2. Digit Selector (DS), X'20' (DS notated **d**)
     - If a nonzero data digit has been processed previously, or the SI is 1, or the current digit is nonzero, it is converted to EBCDIC and the SI is set to 1. Otherwise the DS is replaced by the FC.
  3. Digit Selector and Significance Start (SS), X'21' (SS notated **s**)
     - The SI is set to 1; if the current digit is nonzero, it is converted to EBCDIC. Otherwise the SS is replaced by the FC.
  4. Field Separator (FS), X'22' (FS notated **f**)
     - The SI is reset to 0, and the FS is replaced by the FC.
  5. Message character having any other value; unchanged or replaced by FC
     - Things like decimal points, currency signs, +/− signs, and text like CREDIT
- A pattern like X'402020204B202120' is represented by C'•ddd,dsd'

- Each edit step produces one of three results, in this priority:
  1. A zoned source digit replaces a DS or SS in the pattern
     If: the digit is nonzero, or the SI is ON
  2. The FC replaces the pattern character
     If: the SI is OFF, or the pattern character is FS
  3. The pattern character is unchanged
     If: the SI is ON, or the pattern character is the FC
- SI settings:

  OFF: (1) at start, (2) after FS, (3) source byte has + code in rightmost digit

  ON: if no + code in rightmost digit, then (1) SS and valid digit, (2) DS and nonzero digit
- CC settings:

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)

1. A small number

```
            MVC   PgNum,PgNPat       Copy pattern to result area
            ED    PgNum(4),PgNo      Convert to characters
            – – –
PgNo        DC    PL2'7'             Page number 007+
PgNum       DS    CL4                Edited result = C'•••7'
PgNPat      DC    C' ',3X'20'        Pattern = C' ddd'
```

- A zero value converts to all blanks!

2. 32-bit binary integer; note SS before last DS

```
            L     0,Num              Get nonnegative binary 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'•ddddddddsd'
LineX       DS    CL12               Edited result here, C'••1234567890'
```

- Editing starts after 4 high-order zero digits; the SS ensures that a zero value displays at least one digit

- Inserting commas in large integer values (see Example 2 on slide 35)

```
            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'  C'•ss,sss,sss,sds' (X'6B' is a comma)
LineX       DS    CL(LineX)          Edited result: C'••1,234,567,890'
```

- Editing negative values (like a credit on a charge-card bill)

```
            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 = 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
```

  - If the balance is -$123.45 (the bank owes you) the result is
    `Your•Account•Balance•is••••123.45•CREDIT`
  - If the balance is +$321.09 (you owe the bank) the result is
    `Your•Account•Balance•is••••321.09•••••••`

- EDMK is identical to ED, except:
  - If the SI is OFF when the first significant digit is zoned into the pattern, its address is put in GR1
- Example: a “floating” currency symbol

```
         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'
```

- If the first significant digit is forced by a SS, the SI will be ON and GR1 remains unchanged

- One execution of ED/EDMK can edit multiple fields
  - A field separator (FS) (1) sets the SI OFF, and (2) is replaced by the FC
- Example: edit two packed decimal values

```
            ED     Pat2,PD2              Edit two packed decimal values
            - - -
PD2         DC     P'+024',P'-135'       Two values
Pat2        DC     X'40202120402220212O'  C'•dsdf•dsd'
```

  - The result is C'••24••135'
- With EDMK, if more than one nonzero digit forces the SI ON, only the address of the rightmost is placed in GR1
  - The SI will then be OFF if that digit has a + code in the right digit

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

- These are complex but powerful instructions

- We use some simple macro instructions for input, output, conversion, and display

  `CONVERTI` converts decimal characters in memory to 32- or 64-bit binary integers in a general register

  `CONVERTO` converts 32- or 64-bit binary integers in a general register to decimal characters, or contents of a floating-point register to hexadecimal characters, in memory

  `DUMPOUT` displays the contents of storage in hexadecimal and character formats

  `PRINTLIN` sends a string of characters to a printer file

  `PRINTOUT` displays the contents of registers and of named areas of memory, and/or terminates execution

  `READCARD` reads an 80-byte record from an input file to a specified area of memory

- Each macro calls an entry point in an automatically generated control section

- The macro descriptions use these terms:

`<name>`	a symbol naming an area of memory addressable from the macro
`<number>`	a self-defining term (or a predefined absolute symbol) with value limits specified by the macro
`<d(b)>`	specifies an addressable base-displacement operand
`<address>`	specifies a `<name>` or `<d(b)>`
`<nfs>`	an optional name-field symbol on a macro
`[item]`	[ ] indicates an optional item
`...`	indicates that the preceding item may be repeated

- Referring to registers:
  - Numbers 0-15 refer to 32-bit general registers 0-15
  - Numbers 16-31 refer to 64-bit general registers 0-15
  - Numbers 32-47 refer to Floating-Point registers 0-15

- CONVERTI is written

```
<nfs>    CONVERTI  <number>,<address>[,ERR=<address>][,STOP=<address>]
```

  - The digits starting at the second operand <address> are converted to binary in the general register designated by <number>
    - The first non-blank character must be +, −, or a decimal digit; if not, GR1 is set to the address of the invalid character
  - ERR= specifies an address to receive control for an invalid <number> or a too-large converted value
  - STOP= specifies an address to receive control for an invalid character in the input
  - If either condition occurs and neither ERR= or STOP= is specified, the program terminates with an error message.

- Example:

```
         CONVERTI  3,Data    c(GR3) = X'00000013', c(GR1) = A(Data+3)  ('?')
         - - -
Data     DC    C'+019?'      Input string
```


- CONVERTO is written

```
<nfs>    CONVERTO  <number>,<address>
```

- The contents of the register specified by <number> (*not* the <number> itself!) are converted to N characters in memory starting at <address>

  0 ≤ <number> ≤ 15: N=12
  16 ≤ <number> ≤ 31: N=21
  32 ≤ <number> ≤ 47: N=20

- The first character of the result is always a blank
- If the value of <number> is not between 0 and 47, the macro is ignored
- Converted negative binary values are preceded with a – sign
- Examples (where • represents a blank character):

```
CONVERTO  4,GR4Val     c(GR4Val) = •–2147483648
CONVERTO  22,GG6Val    c(GG6Val) = •–9223372036854775808
CONVERTO  34,FR2Val    c(FR2Val) = •X'FEDCBA9876543210'
```


- DUMPOUT prints a formatted display of memory (a “dump”)

```
<nfs>    DUMPOUT   <address>[,<address>]
```

  - If only one operand is present, only one line is dumped
  - If both operands are present, the dump is from the lower address to the higher
- Each line starts on a word boundary and displays 32 bytes
  - The first line contains the byte at the lower address
  - The last line contains the byte at the higher address
- Example:

```
          Dumpout  A,B        Dump,including bytes from A to B
```

  - produces something like this:

```
***  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.....*
```


- PRINTLIN sends up to 121 characters to a print file

```
<nfs>    PRINTLIN  <address>[,<number>]
```

  - The character string starts at <address>
  - <number> is the number of characters (at most 121)
    - If <number> is omitted, it is assumed to be 121

- The first character is used for vertical spacing ("carriage control") and is not printed:
  - EBCDIC ' ' (blank) means single space
  - EBCDIC '0' (zero) means double space
  - EBCDIC '-' (minus) means triple space
  - EBCDIC '1' (one) means start at the top of a new page
  - EBCDIC '+' (plus) means *no* spacing

- Example:

```
PrTtl    PRINTLIN  Title
         _ _ _
Title    DC   CL121'1Title for Top Line of a Page'
```


- PRINTOUT supports 3 types of operand: <name>, <number>, and *

```
<nfs>    PRINTOUT  [<name>,...][<number>,...]

<nfs>    PRINTOUT  *
```

  - * terminates execution; it is treated as the last operand
  - a <number> operand between 0 and 47 refers to a register; other values are treated as an address, or ignored
  - a <name> operand causes the named area to be printed; format and length depend on operand attributes

- Examples:

```
PrintOut 1,19,32,*       Print GR1, GG3, FPR0, terminate
```

- Produces output like this:

```
*** 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
```


- READCARD reads 80-byte records into your program

```
<nfs>    READCARD  <address>[,<address>]
```

  - The first operand specifies the location in your program for the record
  - If no records remain (“end of file”, EOF)
    1. If the second operand is present, control is returned to that location
    2. If the second operand is omitted, the program is terminated with a message

```
*** Execution terminated by Reader EOF
```

- Example:

```
GetARec  READCARD  MyRecord,EndFile
         - - -
EndFile  - - -     Do something about no more records
```


- Previous examples (slides 5 and 7) have illustrated DUMPOUT/PRINTOUT header lines:

```
   *** 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
  - where CC=n is the Condition Code at that point

- The header line can be suppressed by specifying an operand

```
Header=no
```

  at any position in the operand list; mixed case is OK

- Examples:

```
DUMPOUT  A,B,Header=NO
PRINTOUT 0,19,header=no,34,*
```


- All the macros must execute in 24-bit addressing mode, AMODE(24), and reside below the 16MB “line”, RMODE(24)
- The instructions generated by the macros are self-modifying, as is the generated “service” control section; programs using these macros are not reenterable
- Most operands of the form `<address>`, `<name>`, and `<number>` are resolved in S-type address constants, so addressability is required when a macro is invoked
- Be careful not to reference areas outside your program
- At most 8 characters of `<name>` and `<d(b)>` operands are displayed by `PRINTOUT`

- This sample program and its listing and output are shown in the text

```
         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
```

