Assembler Language Programming

for

IBM System z™ Servers

Mini-Slides and Lecturer Notes, Version 2.00

Chapters I to VIII

John R. Ehrman

IBM Silicon Valley Lab
These pages have space below the miniature copies of the lecture slides where a lecturer can write notes for added explanations, digressions, etc.

Because the font sizes used for the full-size slides do not scale exactly to the miniature forms, some of the miniature slides might overflow the space in which the full-size slides will fit. I expect that these miniature slides will be used only to connect the full-size lecture slides to these lecturer notes, so the overflows won’t be a major concern.
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

### Notes

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

```
 ┌────────┬────────┐
 │ Field1 │ Field2 │
 └────────┴────────┘
```

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 $B_j$, 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

### Notes
What’s an “Operand”? 3

• 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

\[ \text{LM } R_1, R_3, D_2 (B_2) \]

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:

\[ \text{LM } 2, 12, \text{SaveArea} \]

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:

\[ \text{LM } 2, 12, \text{SaveArea} \]

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

Notes

Binary and Hexadecimal Numbers 4

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

Notes
Positional Notation and Binary Numbers

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

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

Hexadecimal Numbers

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

<table>
<thead>
<tr>
<th>Binary (bin)</th>
<th>Decimal (dec)</th>
<th>Hexadecimal (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>

- 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’
Converting Between Bases

- 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

Notes

Number Representations: Unsigned and Signed

- 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

Notes
Two's Complement

• Binary addition is very simple:
  \[
  \begin{array}{cccc}
  \ 0 & 0 & 1 & 1 \\
  +0 & +1 & +0 & +1 \\
  \hline
  \ 0 & 1 & 1 & 0 \quad \text{(carry)}
  \end{array}
  \]

• 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 11000000 ones' complement
    \(+1\) \(+1\)
    00000001 (signed \(+3\)) 11100000 (signed \(-31\))

  - Carries out of the leftmost bit: ignored for unsigned, important for signed

  * But most arithmetic instructions take note of carries

Sign Extension

• 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
Addition, Subtraction, and Arithmetic Overflow

• All bits are added; high-order carries are lost (but noted)
  - Examples, using signed 4-bit values (range -8 ≤ value ≤ +7):
    
    | Value | 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):
    
    | Value | 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 one’s 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:
  
<table>
<thead>
<tr>
<th>Value</th>
<th>1111 (-1)</th>
<th>+2 (+2)</th>
<th>3-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0010 (+2)</td>
<td>0011 (+3)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>1110 (+1, comp)</td>
<td>0101 (-2, comp)</td>
<td>1010 (+5, comp)</td>
</tr>
<tr>
<td>+ 1</td>
<td>1110 (-2, carry)</td>
<td>+0000 (+0, carry)</td>
<td>+110 (-2, no carry)</td>
</tr>
</tbody>
</table>

- 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!
Addition, Subtraction, and Arithmetic Overflow ...

- 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, 1's 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

Notes

A Circular Representation of 4-bit Signed Integers

- A circular representation of 4-bit signed integers:

  0 = 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

Notes
• 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:

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

Notes

Exercise Answers

• Slide 5:
  - B'01010001', B'11111111'
  - 170, 241

• Slide 6:
  - X'91', X'1F4'
  - 1891, 247
  - B'11101100011', B'111101111'

• Slide 7:
  - X'929', 6560, 10B513

• Slide 8:
  - −86, 170

Notes
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 Organization

• 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:

```
<table>
<thead>
<tr>
<th>BDF</th>
<th>BE0</th>
<th>BE1</th>
<th>BE2</th>
<th>BE3</th>
<th>BE4</th>
<th>BE5</th>
<th>BE6</th>
<th>BE7</th>
<th>BE8</th>
<th>BE9</th>
<th>BEA</th>
<th>BEB</th>
<th>BEC</th>
<th>BED</th>
<th>BEE</th>
<th>BEF</th>
<th>BFD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- HALFWORD | HALFWORD | HALFWORD | HALFWORD | HALFWORD | HALFWORD | HALFWORD | HALFWORD | HALFWORD
- WORD     | WORD     | WORD     | WORD     | WORD     | WORD     | WORD     | WORD     |
- DOUBLEWORD | DOUBLEWORD | DOUBLEWORD | DOUBLEWORD | DOUBLEWORD | DOUBLEWORD |
- QUADWORD | QUADWORD |

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

Notes

CPU: General Registers

• 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

```
<table>
<thead>
<tr>
<th>General Register 0</th>
<th>General Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Register 2</td>
<td>General Register 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>General Register 14</td>
<td>General Register 15</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Notes
CPU: Floating-Point Registers and PSW

- 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)

Basic Instruction Cycle

- 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!
Basic Instruction Types

- 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

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Data Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>OPCODE, REGS</td>
</tr>
<tr>
<td>RX</td>
<td>OPCODE, REGS, ADDRESSING HALFWORD</td>
</tr>
<tr>
<td>RS</td>
<td>OPCODE, REGS, ADDRESSING HALFWORD</td>
</tr>
<tr>
<td>SS</td>
<td>OPCODE, LENGTHS, ADDRESSING HALFWORD, ADDRESSING HALFWORD</td>
</tr>
</tbody>
</table>

Notes

Instruction Lengths

- 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)

<table>
<thead>
<tr>
<th>ILC (decimal)</th>
<th>ILC (binary)</th>
<th>Instruction types</th>
<th>Opcode bits 0-1</th>
<th>Instruction length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B’00’</td>
<td>Not available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B’01’</td>
<td>RR</td>
<td>B’00’</td>
<td>One halfword</td>
</tr>
<tr>
<td>2</td>
<td>B’10’</td>
<td>RX</td>
<td>B’01’</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>2</td>
<td>B’10’</td>
<td>RS, SI</td>
<td>B’10’</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>3</td>
<td>B’11’</td>
<td>SS</td>
<td>B’11’</td>
<td>Three halfwords</td>
</tr>
</tbody>
</table>

Notes
Interruptions

• The basic instruction cycle is modified to handle interruptions:

![Diagram of instruction cycle with 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)

Notes

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

• 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.

Notes
Addressing and Address Generation

- 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 is not zero, add c(Rb); ignore carries.
  2b. If the base digit 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.
    - Step 2b: C73E9384 Effective Address.

Notes

Indexing and Virtual Addresses

- 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)CE184EIC 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.

Notes
Chapter III. Assembler Language Programs

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.

Notes

Assembler Language

- 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

Notes
Processing Your Program

- 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

---

Preparing Assembler Language Statements

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

```
+------------+------------+---------------------+
|            |            |                     |
|            |            |                     |
| first char | last char  |                     |
+------------+------------+---------------------+
|  1         |  10        |  20                 |
|  30        |  40        |  50                 |
|  60        |  70        |  80                 |
|  ....v....|  ....v....|  ....v....          |
|  ....v....|  ....v....|  ....v....          |
|  ....v....|  ....v....|  ....v....          |
|  ....v....|  ....v....|  ....v....          |
```

- 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
Statement Fields

- 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)

Writing Simple Programs

- 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!
A Sample Program

- An example of a small program with "Job Control" statements

```
//JRETEST JOB (A925,2236067977), 'J.EHRMAN'
EXEC ASMACLG
//C.SYSIN DD *
Test Start 0 First line of program
Print NoGen
Sample Program
BASR 15,0 Establish a base register
Using ",,15 Inform the Assembler
PRINTOUT MyName," Print name and stop
MyName DC 'John R. Ehrman' Define constant with name
END Test Last statement
/*
```

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

Basic Macro Instructions

- 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.
Self-Defining Terms and Symbols

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

Self-Defining Terms

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 $2^{31} - 1$ (2,147,483,647)
- **Hexadecimal**: a string of hexadecimal digits enclosed in ‘X...X’ with value between ‘X0’ and ‘XFFFFFFFF’ ($2^{32} - 1$, 4,294,967,295)
- **Binary**: a string of binary digits enclosed in ‘B...B’ with value between ‘B0’ and ‘B00000000000000000000000000000000’ ($2^{32} - 1$, 4,294,967,295)
- **Character**: a string of 1-4 characters enclosed in ‘C...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
EBCDIC Character Representation

• Every character is represented by a number

<table>
<thead>
<tr>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>40</td>
<td>c</td>
<td>63</td>
<td>g</td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>d</td>
<td>64</td>
<td>h</td>
<td>68</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>e</td>
<td>65</td>
<td>i</td>
<td>69</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>43</td>
<td>f</td>
<td>66</td>
<td>j</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>g</td>
<td>67</td>
<td>k</td>
<td>71</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>h</td>
<td>68</td>
<td>l</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>46</td>
<td>i</td>
<td>69</td>
<td>m</td>
<td>73</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>47</td>
<td>j</td>
<td>70</td>
<td>n</td>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>k</td>
<td>71</td>
<td>o</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>l</td>
<td>72</td>
<td>p</td>
<td>76</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>m</td>
<td>73</td>
<td>q</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>n</td>
<td>74</td>
<td>r</td>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>o</td>
<td>75</td>
<td>s</td>
<td>79</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>p</td>
<td>76</td>
<td>t</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>54</td>
<td>q</td>
<td>77</td>
<td>u</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>r</td>
<td>78</td>
<td>v</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>s</td>
<td>79</td>
<td>w</td>
<td>83</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>57</td>
<td>t</td>
<td>80</td>
<td>x</td>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>u</td>
<td>81</td>
<td>y</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>v</td>
<td>82</td>
<td>z</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>w</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

Symbols and Attributes

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!)
Program Relocatability

- 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 Location Counter

- 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
Assigning Values to Symbols

- 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 'A'               Char_A has value 'A'
     ```

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

Symbols and Variables

- 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

Notes
• 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)

Terms

• 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)
Expressions

- An expression is an arithmetic combination of terms and operators
  $7 + 4 \times X - N/L \times \text{Item Size} \times \text{Count}$
- A parenthesized expression is treated as a term
  $(A + 2) \times (X \times 4780 - 33) + (7) + (6 - 2)$
  - Parenthesized sub-expressions are evaluated first
- Unary (prefix) $+$ and $-$ are allowed

Notes

Evaluating Assembly-Time Expressions

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, $2 + 5 \times 3 - 6 \rightarrow (2 + (5 \times 3)) - 6$, not $((2 + 5) \times 3) - 6$
4. No relocatable terms allowed in multiplication or division
5. For same-priority operations, evaluate from left to right
   So, $5 \times 2 / 4 \rightarrow (5 \times 2) / 4, 5 / 2 \times 4 \rightarrow (5 / 2) \times 4$
6. Multiplication retains low-order 32 bits of 64-bit product

Notes
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)

Notes

Machine Instruction Statement Operand Formats

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

<table>
<thead>
<tr>
<th>expr</th>
<th>expr₁(expr₆)</th>
<th>expr₁(expr₂,expr₃)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8(7)</td>
<td>22(22,22)</td>
</tr>
<tr>
<td>8*4N</td>
<td>4(8)</td>
<td>4(4(8),(8),(C))</td>
</tr>
<tr>
<td>(91)</td>
<td>(91),(15)</td>
<td>(91),(15)</td>
</tr>
</tbody>
</table>

- In the second and third forms, adjacent parentheses do not imply multiplication!
- In the third form, expr₂ can be omitted if it is zero:
  expr₁(expr₆) [The comma is still required!]
Instructions, Mnemonics, Operands

- 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
    - expr₁ — absolute or relocatable
    - expr₁(expr₂) — expr₁ absolute or relocatable, expr₂ absolute
    - expr₁(expr₂,expr₃) — all three expr's absolute
  - Explicit and implied addresses
  - Explicit and implied lengths

Basic RR-Type Instructions

- These are some commonly used RR-type instructions:

<table>
<thead>
<tr>
<th>Op</th>
<th>Mнем</th>
<th>Instruction</th>
<th>Op</th>
<th>Mнем</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>BALR</td>
<td>Branch And Link</td>
<td>06</td>
<td>BCTR</td>
<td>Branch On Count</td>
</tr>
<tr>
<td>07</td>
<td>BCR</td>
<td>Branch On Condition</td>
<td>0D</td>
<td>BASR</td>
<td>Branch And Save</td>
</tr>
<tr>
<td>10</td>
<td>LPR</td>
<td>Load Positive</td>
<td>11</td>
<td>LNR</td>
<td>Load Negative</td>
</tr>
<tr>
<td>12</td>
<td>LTR</td>
<td>Load And Test</td>
<td>13</td>
<td>LCR</td>
<td>Load Complement</td>
</tr>
<tr>
<td>14</td>
<td>NR</td>
<td>AND</td>
<td>15</td>
<td>CLR</td>
<td>Compare Logical</td>
</tr>
<tr>
<td>16</td>
<td>OR</td>
<td>OR</td>
<td>17</td>
<td>XR</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>18</td>
<td>LR</td>
<td>Load</td>
<td>19</td>
<td>CR</td>
<td>Compare</td>
</tr>
<tr>
<td>1A</td>
<td>AR</td>
<td>Add</td>
<td>1B</td>
<td>SR</td>
<td>Subtract</td>
</tr>
<tr>
<td>1C</td>
<td>MR</td>
<td>Multiply</td>
<td>1D</td>
<td>DR</td>
<td>Divide</td>
</tr>
<tr>
<td>1E</td>
<td>ALR</td>
<td>Add Logical</td>
<td>1F</td>
<td>SLR</td>
<td>Subtract Logical</td>
</tr>
</tbody>
</table>

- Typical operand field described as R₁,R₂ — operands of "expr₁" form
• Assembler must generate machine language form of the instruction:

```
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
</table>
```

• R1 and R2 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

---

**Basic RX-Type Instructions**

- These are some commonly used RX-type instructions:

<table>
<thead>
<tr>
<th>Op</th>
<th>Mнем</th>
<th>Instruction</th>
<th>Op</th>
<th>Mнем</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>STC</td>
<td>Store Character</td>
<td>43</td>
<td>IC</td>
<td>Insert Character</td>
</tr>
<tr>
<td>44</td>
<td>EX</td>
<td>Execute</td>
<td>45</td>
<td>BAL</td>
<td>Branch And Link</td>
</tr>
<tr>
<td>46</td>
<td>BCT</td>
<td>Branch On Count</td>
<td>47</td>
<td>BC</td>
<td>Branch On Condition</td>
</tr>
<tr>
<td>4D</td>
<td>BAS</td>
<td>Branch And Save</td>
<td>50</td>
<td>ST</td>
<td>Store</td>
</tr>
<tr>
<td>54</td>
<td>N</td>
<td>AND</td>
<td>55</td>
<td>CL</td>
<td>Compare Logical</td>
</tr>
<tr>
<td>56</td>
<td>O</td>
<td>OR</td>
<td>57</td>
<td>X</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>58</td>
<td>L</td>
<td>Load</td>
<td>59</td>
<td>C</td>
<td>Compare</td>
</tr>
<tr>
<td>5A</td>
<td>A</td>
<td>Add</td>
<td>5B</td>
<td>S</td>
<td>Subtract</td>
</tr>
<tr>
<td>5C</td>
<td>M</td>
<td>Multiply</td>
<td>5D</td>
<td>D</td>
<td>Divide</td>
</tr>
<tr>
<td>5E</td>
<td>AL</td>
<td>Add Logical</td>
<td>5F</td>
<td>SL</td>
<td>Subtract Logical</td>
</tr>
</tbody>
</table>

- RX-instruction first operand field described as R1 ("expr1" form)
- Second operand field described as S2 ("expr2" form), as S2(X2) ("expr2(expr1)" form), as D2(X2,B2) ("expr2(expr1,expr2)" form), or as D2(B2) ("expr2(expr1)" form)
Writing RX-Type Instructions

- Assembler must generate machine language form of the instruction:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
</table>

- First operand designates a general register
- Second operand usually designates a memory reference
  - B2, D2, and X2 components used at execution time to calculate memory address (as described in Section 5)
  - Generic RX-instruction operands: `Rn, address-specification`
- Since L opcode is `X’58’`,
  - `L 1,200(9,12)` will generate
    | 58 | 1 | 9 | C | 0c8 |
  - `L 1,200(,12)` will generate
    | 58 | 1 | 0 | C | 0c8 |

Notes

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 D2(X2,B2) or D2(B2)
  2. Implicit: The Assembler assigns the base register and displacement for you
     - You specify an operand of the form S2 or S2(X2); the Assembler does assembly-time address resolution (described in Section 10)

- Examples of explicit addresses:
  43047468 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:

<table>
<thead>
<tr>
<th>Explicit Address</th>
<th>Implied Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Indexed</td>
<td>D2(B2)</td>
</tr>
<tr>
<td>Indexed</td>
<td>D2(X2,B2)</td>
</tr>
</tbody>
</table>

Notes
Typical RS- and SI-Type Instructions

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

<table>
<thead>
<tr>
<th>Op</th>
<th>Mnem Type</th>
<th>Instruction</th>
<th>Op</th>
<th>Mnem Type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>STM</td>
<td>RS Store Multiple</td>
<td>91</td>
<td>TM</td>
<td>SI Test Under Mask</td>
</tr>
<tr>
<td>92</td>
<td>MVI</td>
<td>SI Move Immediate</td>
<td>94</td>
<td>NI</td>
<td>SI AND Immediate</td>
</tr>
<tr>
<td>95</td>
<td>CLI</td>
<td>SI Compare Logical Immediate</td>
<td>96</td>
<td>OI</td>
<td>SI OR Immediate</td>
</tr>
<tr>
<td>97</td>
<td>XI</td>
<td>SI Exclusive OR Immediate</td>
<td>98</td>
<td>LM</td>
<td>RS Load Multiple</td>
</tr>
<tr>
<td>88</td>
<td>SRL</td>
<td>RS Shift Right Single Logical</td>
<td>89</td>
<td>SLL</td>
<td>RS Shift Left Single Logical</td>
</tr>
<tr>
<td>8A</td>
<td>SRA</td>
<td>RS Shift Right Single</td>
<td>8B</td>
<td>SLA</td>
<td>RS Shift Left Single</td>
</tr>
<tr>
<td>8C</td>
<td>SRDL</td>
<td>RS Shift Right Double Logical</td>
<td>8D</td>
<td>SDL</td>
<td>RS Shift Left Double Logical</td>
</tr>
<tr>
<td>8E</td>
<td>SRDA</td>
<td>RS Shift Right Double</td>
<td>8F</td>
<td>SLDA</td>
<td>RS Shift Left Double</td>
</tr>
</tbody>
</table>

Many ways to write their operand fields!

Writing RS-Type Instructions

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:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
</table>

$R_1$ operand designates a general register; $R_1$ 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)
31 Writing SI-Type Instructions

- Assembler must generate machine language form of the instruction:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>I2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
</table>

- D1(B1) (first) operand designates an *address-specification*
- Second (I2) operand is an *immediate* operand
- General forms of SI-instruction operands:

<table>
<thead>
<tr>
<th></th>
<th>Explicit Address</th>
<th>Implied Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>D1(B1),I2</td>
<td>S1,I2</td>
</tr>
</tbody>
</table>

- Examples of SI-type instructions

  - MVI 0(6),C
  - CLI Buffer,C

**Notes**

**Typical SS-Type Instructions**

- 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

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

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

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

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>L1</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One Length Field ("SS-1")

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>L1</th>
<th>L2</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two Length Fields ("SS-2")

Addresses and lengths can both be specified explicitly or implicitly.

**SS-1 Form**
- **Explicit Addresses**
  - Explicit Length: \( D_1(N_1, B_1), D_2(B_2) \)
  - Implied Length: \( D_1(B_1), D_2(B_2) \)

- **Implied Addresses**
  - Explicit Length: \( S_1(N_1), S_2(B_2) \)
  - Implied Length: \( S_1, S_2 \)

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

**SS-2 Form**
- **Explicit Addresses**
  - Explicit Lengths: \( D_1(N_1, B_1), D_2(N_2, B_2) \)
  - Implied Lengths: \( D_1(B_1), D_2(B_2) \)

- **Implied Addresses**
  - Explicit Lengths: \( S_1(N_1), S_2(B_2) \)
  - Implied Lengths: \( S_1, S_2 \)

Some examples of SS-2 form instructions:
- PACK 0(8,4),40(5,9) Explicit lengths and addresses
- ZAP Sum(14),.01dSum(4) Explicit lengths, implied addresses
- AP Total(15),.12dNum(12) Implied lengths, explicit addresses
- UNPK String,Data Implied lengths and addresses

Notes
Establishing and Maintaining Addressability

- 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!

The BASR Instruction

- A common method for establishing execution-time addressability uses the RR-type BASR ("Branch and Save") instruction
  
  BASR R₁₄, R₂

  - BASR puts the Instruction Address (IA) in the PSW into the R₁ 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 R₂ operand is zero, nothing more is done
- The address in R₁ can be used as a base address
- R₁ 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 BASH  6,0  ESTABLISH BASE ADDRESS
5002 BEGIN L  2,N  LOAD CONTENTS OF N INTO GR2
5006 A  2,ONE  STORE CONTENTS OF ONE
5008 ST  2,N  STORE CONTENTS OF GR2 INTO N
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...

Notes

---

Computing Displacements (continued)... 38

• Suppose this fragment is to be loaded into memory at address X’84E8’

```
84E8 BASH  6,0  ESTABLISH BASE ADDRESS
84EA BEGIN L  2,N  LOAD CONTENTS OF N INTO GR2
84EE A  2,ONE  STORE CONTENTS OF ONE
84F2 ST  2,N  STORE CONTENTS OF GR2 INTO N
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      0060
84EA      8A06022
84EE      8A06026
84F2      5026022
850C      00000008
8510      00000001
```

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

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>NAME</th>
<th>OPERATION</th>
<th>OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>BEGIN</td>
<td>BASR 6,0</td>
<td>2,8′022′(0,6)</td>
</tr>
<tr>
<td>0002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,8′022′(0,6)</td>
</tr>
<tr>
<td>0004</td>
<td>A</td>
<td>A</td>
<td>2,8′026′(0,6)</td>
</tr>
<tr>
<td>0006</td>
<td>ST</td>
<td>ST</td>
<td>2,8′022′(0,6)</td>
</tr>
</tbody>
</table>

0006 ─ 22 BYTES ─ 0024

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

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>NAME</th>
<th>OPERATION</th>
<th>OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>BEGIN</td>
<td>BASR 6,0</td>
<td>2,8′022′(0,6)</td>
</tr>
<tr>
<td>0002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,N-BEGIN(0,6)</td>
</tr>
<tr>
<td>0004</td>
<td>A</td>
<td>A</td>
<td>2,ONE-BEGIN(0,6)</td>
</tr>
<tr>
<td>0006</td>
<td>ST</td>
<td>ST</td>
<td>2,N-BEGIN(0,6)</td>
</tr>
</tbody>
</table>

0006 ─ 22 BYTES ─ 0024

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

The USING Assembler Instruction and Implied Addresses

• 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:

USING BEGIN,6 GR6 will hold execution address of BEGIN
BASR 6,0 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

### Destroying Base Registers

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Object Code</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0060</td>
<td>BASR 6,0</td>
</tr>
<tr>
<td>0002</td>
<td>58606022</td>
<td>USING BEGIN,6</td>
</tr>
<tr>
<td>0006</td>
<td>5A206026</td>
<td>BEGIN L 6,N ←Wrong register! (6, not 2)</td>
</tr>
<tr>
<td>000A</td>
<td>50206022</td>
<td>ST 2,N</td>
</tr>
<tr>
<td>0024</td>
<td>00000008</td>
<td>N DC F'8'</td>
</tr>
<tr>
<td>0028</td>
<td>00000001</td>
<td>ONE DC F'1'</td>
</tr>
</tbody>
</table>

- 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'000005002'
- When the L instruction is executed, c(GR6) = X'00000000008'
- When the A instruction is executed, its Effective Address is X'00000002028'
  (not X'00000002028'!)
- Worse: the ST will try to store into memory at address X'00000002028', probably causing a memory protection exception

- Be very careful not to alter the contents of base registers!
Calculating Displacements: Assembly Process, Pass 1

- 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

Notes

Calculating Displacements: Assembly Process, Pass 2

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

<table>
<thead>
<tr>
<th>BASEREG</th>
<th>BASE_LOCATION</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>00000002</td>
<td>01</td>
</tr>
</tbody>
</table>

  - For instructions with implied addresses:
    \[ \text{displacement} = (\text{implied_address \ value}) - (\text{base_location \ value}) \]
  - 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:
  \[ \text{Assembly:} \text{displacement} = (\text{operand\_location}) - (\text{base\_location}) \]
  \[ \text{Execution:} (\text{operand\_address}) = \text{displacement} + (\text{base\_address}) \]

Notes
Multiple USING Table Entries

• Suppose there is more than one USING statement:

<table>
<thead>
<tr>
<th>Location</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>BASR 6,0</td>
</tr>
<tr>
<td></td>
<td>USING *,6</td>
</tr>
<tr>
<td>0002</td>
<td>BEGIN L 2,N</td>
</tr>
<tr>
<td></td>
<td>USING *,7</td>
</tr>
</tbody>
</table>

Original USING statement

Added USING statement

• The USING Table now looks like this:

<table>
<thead>
<tr>
<th>BASEREG</th>
<th>BASE_LOCATION</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>00000002</td>
<td>01</td>
</tr>
<tr>
<td>7</td>
<td>00000006</td>
<td>01</td>
</tr>
</tbody>
</table>

− When the A statement is assembled, two addressing halfwords are possible:
  • With register 6: X'00000002'−X'00000002' = X'026' with addressing halfword X'6026'
  • With register 7: X'00000002'−X'00000006' = X'022' with addressing halfword X'7022'

− The Assembler chooses the resolution with the smallest displacement

The DROP Assembler Instruction

• 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:

<table>
<thead>
<tr>
<th>BASEREG</th>
<th>BASE_LOCATION</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>00000006</td>
<td>01</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>BASEREG</th>
<th>BASE_LOCATION</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPTY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMPTY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Addressability Errors

- 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’0000’) and 4095 (X’FFFF’)
     - References requiring displacements outside that range are not addressable
       
       | SetBase | BASR  | USING | L  | 0, | 0+5000 | Would require positive displacement X’1388’ |
       |----------|-------|-------|----|----|---------|------------------------------------------|
       |          | 6, 0  | * ,6  |    |    |         | 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

Notes

Resolutions with Register Zero

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

<table>
<thead>
<tr>
<th>BASEREG</th>
<th>BASE_LOCATION</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>ETC.</td>
<td></td>
</tr>
</tbody>
</table>

- 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’3EB’
  - Two resolutions are possible: addressing halfwords X’03EB’ and X’9258’
    - The Assembler chooses the one with the smaller displacement: X’9258’

Notes
Summary of USING Resolution Rules

- 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.

Notes

Chapter IV: Defining Constants and Storage Areas

- 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.

Notes
Defining Constants

- 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!

Notes

Defining Constants: Basic Types

- 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.

Notes
DC Instruction Statements and Operands

- 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

Boundary Alignment

- 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

Notes
Length Modifiers

- 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:
  
  \[ L_n \text{ or } L(\text{expr}) \]

- Examples:
  
  A DC FL3′8′ Generates X'00000008' at current location
  B DC FL(2*4-5)′8′ Generates X'00000008' 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

Notes

Duplication Factors and Multiple Operands

- 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

Notes
Multiple Nominal Values

- 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,E' 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 89'A' 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

Length Attributes

- 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
• 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       | FE$'$1,2,3,4$'$   | Constants for 100, 200, 300, 400 |
| Hundreds DC       | FE$'$1E1,2E2,3,4E1$'$ | Constants for 100, 2000, 30, 400 |

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

Notes

---

### Seven Basic 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

Notes
F-Type and H-Type Constants

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

<table>
<thead>
<tr>
<th>DC Type</th>
<th>Nominal Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>F'–10'</td>
<td>Generates X'FFFFFFF6', word aligned</td>
<td></td>
</tr>
<tr>
<td>H'–10'</td>
<td>Generates X'FFFF6', halfword aligned</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DC Type</th>
<th>Nominal Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL5’–10'</td>
<td>Generates X'FFFFFFFFF6', unaligned</td>
<td></td>
</tr>
<tr>
<td>HL5’–10'</td>
<td>Generates X'FFFFFFFFF6', unaligned</td>
<td></td>
</tr>
</tbody>
</table>

- 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 Type  | Nominal Value | Result          |
    |----------|---------------|-----------------|
    | F'U2147483648' | Generates X'80000000', word aligned | (2^31) |
    | H'U65535'   | Generates X'FFFF', halfword aligned | (2^16-1) |
    | F'U4294967295' | Generates X'FFFFFFFF', word aligned | (2^32-1) |
    | H'1,U2'    | Generates X'00010002', Mixed forms |

A-Type and Y-Type Address Constants

- 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   | Expr1 | Result          |
    |------|-------|-----------------|
    | Equ 7 | A(C’A’+48) | Generates X'0000000F1', word aligned |
    | DC   | A(R7) | Generates X'0000000F7', word aligned |
    | DC   | A(Expr1) | Will contain execution-time address of Expr1 |
    | DC   | AL1(*─Expr1) | Will generate X'0DC', 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

<table>
<thead>
<tr>
<th>Expr1</th>
<th>DC</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Y(C’A’+48)</td>
<td>Generates X'00F1', halfword aligned</td>
</tr>
<tr>
<td>DC</td>
<td>Y(R7)</td>
<td>Generates X'00F7', halfword aligned</td>
</tr>
</tbody>
</table>

- Early (and very small) machines used relocatable 16-bit address constants
- The ability to generate constants from expressions is very powerful
**Constants of Types C, B, and X**

- 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)

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td><code>DC '12345'</code></td>
<td>Generates X'C12345'; LA of A = 5</td>
</tr>
<tr>
<td>B</td>
<td><code>DC '123456'</code></td>
<td>Generates X'C123456'; LA of B = 3</td>
</tr>
<tr>
<td>C</td>
<td><code>DC 2'B10100101'</code></td>
<td>Generates X'C123456'; LA of C = 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DC ''''</code></td>
<td>Generates X'C'</td>
</tr>
<tr>
<td><code>DC '&amp;&amp;&amp;&amp;'</code></td>
<td>Generates X'C5050'</td>
</tr>
</tbody>
</table>

**Notes**

**Padding and Truncation**

- 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Too Small</th>
<th>Too Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>F,H</td>
<td>Pad with sign bits on left</td>
<td>Truncate on left; error message</td>
</tr>
<tr>
<td>A,Y</td>
<td>Pad with sign bits on left</td>
<td>Truncate on left; error message</td>
</tr>
<tr>
<td>C</td>
<td>Pad with spaces on right</td>
<td>Truncate on right</td>
</tr>
<tr>
<td>B</td>
<td>Pad with zero bits on left</td>
<td>Truncate on left</td>
</tr>
<tr>
<td>X</td>
<td>Pad with zero digits on left</td>
<td>Truncate on left</td>
</tr>
</tbody>
</table>

**Notes**
• A literal is a type of symbol that references and defines a constant

-Written as an equal sign followed by a DC operand

\[ =F'8' =W'22' =X'40' =CL2'S' =A(X'40', C'S') \]

• 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

\[
\begin{align*}
L & Z,=F'8' & \text{Valid reference} \\
ST & Z,=F'8' & \text{Invalid; tries to modify the literal}
\end{align*}
\]

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

---

**The LTORG Assembler Instruction**

- 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
Type Extensions

• 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 'E15' X'0001007EA468000'
  DC AD('C' 'ABC') X'0000000000C12C3'

• 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 C 'ABC' X'C1C2C3' EBCDIC always
  DC C 'ABC' X'414243' ASCII always
  DC C 'ABC' X'004100420043' Unicode UTF-16

Data Storage Definition

• 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
Storage Areas: The DS Assembler Instruction

- 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 F’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

Notes

Zero Duplication Factor

- 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
  - PhoneNumber DS OCL10 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 EQU Assembler Instruction

- 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

The EQU Assembler Instruction, Extended Syntax

- 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' ' Overlay AreaCode
    Prefix Equ AreaCode+3,3,C' ' Overlay Prefix
    Local_No Equ Prefix+3,4,C' ' Overlay Local_No
    ```

Notes
The ORG Assembler Instruction

- The ORG instruction modifies the Location Counter by setting its value to its operand expression:
  \[
  \text{ORG relocatable_expression}
  \]
- 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:
  \[
  \text{ORG relocatable_expression,boundary,offset}
  \]
  - 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 \( \text{+6} \)? Why not \( \text{ORG *,8,-2} \)?

Parameterization

- 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 IntTbls are easier to expand?

Notes

Exercise Solutions

- 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

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.

Notes

General Register Data Transmission

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.

Notes
Load and Store Instructions

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:
  - \( L \ 7,\text{′}F′ \rightarrow \) c(GR7) replaced by X′FFFFFF9F′
  - \( ST \ 7,\text{Num} \) c(NUM) replaced by c(GR7), GR7 unchanged

Notes

Multiple Loads and Stores

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
  - \( L \ 1,A \) and \( ST \ 1,B \)
  - \( L \ 2,A+4 \) and \( ST \ 2,B+4 \)
  - \( L \ 3,A+8 \) and \( ST \ 3,B+8 \)
you can write
  - \( LM \ 1,3,A \) and \( STM \ 1,3,B \)
- The instruction format is
  - \( LM \) (or STM) \( R_i,R_j,S_j \) (explicit address)
  - \( LM \) (or STM) \( R_i,R_j,S_j \) (implied address)
- The register contents are transmitted between GRs and successive words in memory, starting with the \( R_i \) register and ending with the \( R_j \) register
  - If \( R_j \) is smaller than \( R_i \) then GRs \( R_i-15 \) are transmitted followed by GRs \( 0-R_j \)
- These instructions are often used for "status preservation"

Notes
Halfword Data

- 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

  ┌───────────────────┬───────────────────┐
  │ /SM630000 ─ sign─extended /SM630000 ─ │ GR R 1
  └───────────────────┴───────────────────┘

  ┌─────────┴─────────┐
  │ s │ Halfword in memory │
  └───────────────────┘

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

  L 0,=F′65537′ c(GR0)=X′00000001′ +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′00000001′ +65535 = 2^{16}-1
  STH 0,A c(A) = X′FF_FF′ No lost bits, but wrong sign
  LH 1,A c(GR1)=X′00000000′ (-1) Lost significance!

Insert and Store Character

- 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

  ┌─────────────────────────────┬─────────┐
  │ /SM630000 ──────── unchanged ──────── /SM590000 │ │ GR R1
  └─────────────────────────────┴─────────┘

  ┌─────────┐
  │ │ Byte in memory │
  └─────────┘

- Two examples:

  [1] L 0,=F′─1′ c(GR0)=X′00000001′
  IC 0,=C′A′ c(GR1)=X′FF_FF_FF′
  [2] IC 0,X GET 1ST BYTE OF C(X)
  STC 0,=C′A′ STORE AT 2ND BYTE OF Y
  IC 0,X+1 GET 2ND BYTE OF C(X)
  STC 0,=C′A′ STORE BYTE AT Y

  X DS C′A′ c(Y) BECOMES C′BA′

Notes
Insert and Store Multiple Characters

- 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:

  \[
  \text{ICM (or STCM) } R_1, M_3, D_2 \text{ (explicit address)}
  \]
  \[
  \text{ICM (or STCM) } R_1, M_3, S_2 \text{ (implied address)}
  \]

  - 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:

<table>
<thead>
<tr>
<th>CC Meaning</th>
<th>0</th>
<th>M_3 = 0, or all inserted bytes are zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M_3 = 0, or all inserted bytes are zero</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>M_3 = 0, or all inserted bytes are zero</td>
<td></td>
</tr>
</tbody>
</table>

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

  \[
  \text{ICM } 1,\text{’B’0101’,=,c’X’1122’} \quad \text{c(GR1) now = X’AA11CC22’, CC=2}
  \]

  \[
  \text{STCM } 1,\text{’B’1010’,Z} \quad \text{c(Z) now = X’AAC’}
  \]

RR-Type Data Transmission Instructions

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>c(GR R1) /SM630000─ c(GR R2)</td>
<td>Unchanged</td>
</tr>
<tr>
<td>LTR</td>
<td>c(GR R1) /SM630000─ c(GR R2)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCR</td>
<td>c(GR R1) /SM630000─ ─c(GR R2)</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>LPR</td>
<td>c(GR R1) /SM630000─ │c(GR R2)</td>
<td>0,2,3</td>
</tr>
<tr>
<td>LNR</td>
<td>c(GR R1) /SM630000─ ─│c(GR R2)</td>
<td>0,1</td>
</tr>
</tbody>
</table>

- 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

  \[
  \text{LR } 0,1 \quad \text{c(R0) ← c(R1), CC unchanged}
  \]

  \[
  \text{LTR } 1,1 \quad \text{Test c(R1), set CC}
  \]

  \[
  \text{LCR } 2,1 \quad \text{c(R2) = –c(R1), set CC}
  \]

  \[
  \text{LNR } 3,1 \quad \text{c(R2) = –c(R1), set CC}
  \]
Load, Store, and Insert for 64-bit General Registers

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

- Some instructions use all bits of a 64-bit GR:
  - LG $R_1$, $S_2$: Load a doubleword into 64-bit GR
  - STG $R_1$, $S_2$: Store a doubleword from 64-bit GR
  - LMG $R_1$, $R_3$, $S_2$: Load doublewords into 64-bit GRS
  - STMG $R_1$, $R_3$, $S_2$: Store doublewords from 64-bit GRS

- Some instructions use only the leftmost 32 bits of a 64-bit GR:
  - LMH $R_1$, $R_3$, $S_2$: Load words into left halves of 64-bit GRS
  - STMH $R_1$, $R_3$, $S_2$: Store words from left halves of 64-bit GRS
  - ICMH $R_1$, $M_3$, $S_2$: Insert characters into left half of GR
  - STCMH $R_1$, $M_3$, $S_2$: Store characters from left half of GR

- Some of these instructions use RXY or RSY format:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>DL2</th>
<th>DH2</th>
<th>OPCODE</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>DL2</th>
<th>DH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LTG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LCGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LPGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LNGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LNGR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

RR-Type Data Transmission Instructions for 64-bit GPRs

- These instructions are similar to the 32-bit forms, but now have 6 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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGR</td>
<td>$(GG_{R2}) \leftarrow (GG_{R1})$</td>
<td>Not changed</td>
</tr>
<tr>
<td>LTGR</td>
<td>$(GG_{R1}) \leftarrow (GG_{R2})$</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCGR</td>
<td>$(GG_{R1}) \leftarrow - (GG_{R2})$</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>LPGR</td>
<td>$(GG_{R1}) \leftarrow</td>
<td>(GG_{R2})$</td>
</tr>
<tr>
<td>LNGR</td>
<td>$(GG_{R1}) \leftarrow -</td>
<td>(GG_{R2})$</td>
</tr>
</tbody>
</table>

- Examples:
  * Assume $c(GG_2) = +1$, $c(GG_3) = 0$
    - LGR 7,3 $c(GG_7)=0$, CC not set
    - LTGR 2,2 $c(GG_2)=+1$, CC=2
    - LMG 1,3 $c(GG_1)=0$, CC=0
    - LCGR 4,2 $c(GG_4)=-1$, CC=1
    - LPGR 0,4 $c(GG_0)=+1$, CC=2
    - LNGR 5,2 $c(GG_5)=-1$, CC=1
Load and Test Instructions

• These instructions can replace an equivalent pair:
  
  \[
  \begin{align*}
  LT & \quad R_1, S_2 \\
  \text{replaces} & \quad L \quad R_1, S_2 \\
  & \quad \text{LTR} \quad R_1, R_1 \\
  \text{or even} & \quad \text{ICM} \quad R_1, B'1111', S_2 \\
  \text{and,} & \quad \text{LTG} \quad R_1, S_2 \\
  \text{replaces} & \quad \text{LG} \quad R_1, S_2 \\
  & \quad \text{LTGR} \quad R_1, R_1 \\
  \end{align*}
  \]

• 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

Notes

Mixed 32- and 64-bit Operands

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

\[
\begin{array}{ccc}
\text{GG R}_1 & \text{GG R}_1 & \text{GG R}_1 \\
\hline
\text{0} & \text{31} & \text{63} \\
\text{32-bit second operand} & \text{c} & \text{c} \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCFR</td>
<td>c(GG R1) --- c(word in memory)</td>
<td>Not changed</td>
</tr>
<tr>
<td>LCFR</td>
<td>c(GG R1) --- c(GR R2)</td>
<td>Not changed</td>
</tr>
<tr>
<td>LTFR</td>
<td>c(GG R1) --- c(GR R2)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCFR</td>
<td>c(GG R1) --- c(GR R2)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LFGR</td>
<td>c(GG R1) --- c(GR R2)</td>
<td>0,2</td>
</tr>
<tr>
<td>LFGR</td>
<td>c(GG R1) --- c(GR R2)</td>
<td>0,1</td>
</tr>
</tbody>
</table>

• Example

\[
\text{LCFR 0,1 IS EQUIVALENT TO LFGR 0,1} \\
\text{LCFR 0,0} \\
\]

Notes
Other General Register Load Instructions

- These instructions simplify occasional programming tasks:
  - Load Byte: insert a byte into the rightmost Rₙ register byte and sign-extend its leftmost bit (LB, LBR; LGB, LGBR)
  - Load Logical Character: insert a byte into the rightmost Rₙ register byte and zero the remaining register bits (LLC, LLCR; LLGC, LLGCR)
  - Load Logical Halfword: insert 2 bytes into the rightmost Rₙ 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

Easy Misunderstandings

- 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 \textit{not} met:
- Continue execution with the next sequential instruction

If the branch condition \textit{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\textsubscript{R2}
  - But if the R\textsubscript{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 Mask and Branch Condition

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

```
<table>
<thead>
<tr>
<th>07</th>
<th>M1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>M1</td>
<td>X2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- 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

<table>
<thead>
<tr>
<th>CC value tested</th>
<th>Instruction bit</th>
<th>Mask bit position</th>
<th>Mask bit position</th>
<th>Mask bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Examples of Conditional Branch Instructions

1. Branch to XX if the CC is zero.
   \( \text{BC 8,XX} \) \( M_1 = \text{B’1000’} \)

2. Branch to XX if the CC is not 0.
   \( \text{BC 7,XX} \) \( M_1 = \text{B’0111’} \)

3. Always branch to the instruction whose address is contained in GR14.
   \( \text{BCR 15,14} \) \( M_1 = \text{B’1111’} \)
   \( \text{or BC 15,0(0,14) M1 = B’1111’} \)
   - 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.
   \( \text{BC 5,XX} \) \( M_1 = \text{B’0101’} \)

Notes
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:
  
  \[
  \begin{align*}
  &\text{BC} \ 0, x \\
  &\text{BCR} \ 0, \text{any}
  \end{align*}
  \]
- The Assembler provides special "extended mnemonics" for them:
  
  \[
  \begin{align*}
  &\text{NOP} \ S_2 \text{ is equivalent to } \text{BC} \ 0, S_2 \\
  &\text{NOPR} \ R_2 \text{ is equivalent to } \text{BCR} \ 0, R_2
  \end{align*}
  \]
- Some no-operation instructions have special side-effects:
  
  \[
  \text{BCR} \ 15, 0 \text{ ("branch always nowhere")}
  \]
  causes internal overlaps of fetch/decode/execute phases to slow
  - Sometimes helps with problem diagnosis or potential memory conflicts

Notes

Conditional No-Operation

- Some instructions must be aligned on a specific boundary
  - Such as a \text{BASR} \ 14,15 instruction on a halfword boundary before a fullword
- Use the \text{CNOP} ("Conditional No-Operation") instruction to request the desired alignment
  
  \[
  \text{CNOP} \ \text{boundary, width}
  \]
  - \text{width} is either 4, 8, or 16
  - \text{boundary} is an even number such that \text{boundary} < \text{width}
  
  \[
  \begin{align*}
  &\text{CNOP} \ 2, 4 \quad \text{Align to the next halfword before a fullword} \\
  &\text{CNOP} \ 0, 8 \quad \text{Align to the next doubleword boundary}
  \end{align*}
  \]

Notes
### Extended Mnemonics

- Don’t memorize mask-bit positions; use “Extended Mnemonics”

<table>
<thead>
<tr>
<th>RX Mnemonic</th>
<th>RR Mnemonic</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>BR</td>
<td>15</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BNO</td>
<td>BNOR</td>
<td>14</td>
<td>Branch if Not Ones or No Overflow</td>
</tr>
<tr>
<td>BNH</td>
<td>BNHR</td>
<td>13</td>
<td>Branch if Not High</td>
</tr>
<tr>
<td>BNP</td>
<td>BNPR</td>
<td>13</td>
<td>Branch if Not Plus</td>
</tr>
<tr>
<td>BNL</td>
<td>BNLR</td>
<td>11</td>
<td>Branch if Not Low</td>
</tr>
<tr>
<td>BN/M</td>
<td>BNM/R</td>
<td>11</td>
<td>Branch if Not Minus or Not Mixed</td>
</tr>
<tr>
<td>BE</td>
<td>BER</td>
<td>8</td>
<td>Branch if Equal</td>
</tr>
<tr>
<td>BZ</td>
<td>BZR</td>
<td>8</td>
<td>Branch if Zero(s)</td>
</tr>
<tr>
<td>BNE</td>
<td>BNER</td>
<td>7</td>
<td>Branch if Not Equal</td>
</tr>
<tr>
<td>BNZ</td>
<td>BNZR</td>
<td>7</td>
<td>Branch if Not Zero</td>
</tr>
<tr>
<td>BL</td>
<td>BLR</td>
<td>4</td>
<td>Branch if Low</td>
</tr>
<tr>
<td>BM</td>
<td>BMR</td>
<td>4</td>
<td>Branch if Minus, or if Mixed</td>
</tr>
<tr>
<td>BH</td>
<td>BHR</td>
<td>2</td>
<td>Branch if High</td>
</tr>
<tr>
<td>BP</td>
<td>BPR</td>
<td>2</td>
<td>Branch if Plus</td>
</tr>
<tr>
<td>BG</td>
<td>BGR</td>
<td>1</td>
<td>Branch if Ones, or if Overflow</td>
</tr>
<tr>
<td>NOP</td>
<td>NOPR</td>
<td>0</td>
<td>No Operation</td>
</tr>
</tbody>
</table>

### Fixed-Point Binary Basic Arithmetic

- 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
### Signed-Arithmetic Add and Subtract Instructions

- **Instructions with 32-bit operands**

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>Add halfword from memory</td>
<td>SR</td>
<td>Subtract halfword from memory</td>
</tr>
<tr>
<td>A</td>
<td>Add word from memory</td>
<td>S</td>
<td>Subtract word from memory</td>
</tr>
<tr>
<td>AR</td>
<td>Add word from c(GR R₂)</td>
<td>SGR</td>
<td>Subtract word from c(GR R₂)</td>
</tr>
</tbody>
</table>

- **Instructions with 64-bit operands**

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6</td>
<td>Add doubleword from memory</td>
<td>SG</td>
<td>Subtract doubleword from memory</td>
</tr>
<tr>
<td>AGR</td>
<td>Add doubleword from c(GG R₂)</td>
<td>SGR</td>
<td>Subtract doubleword from c(GG R₂)</td>
</tr>
</tbody>
</table>

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

---

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC Setting and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(GR R₁) = c(GR R₁) ± c(GR R₂)</td>
<td>0: Result is zero; no overflow</td>
</tr>
<tr>
<td>c(GR R₁) = c(GR R₁) ± c(word in memory)</td>
<td>1: Result is &lt; zero; no overflow</td>
</tr>
<tr>
<td>c(GG R₁) = c(GG R₁) ± c(GG R₂)</td>
<td>2: Result is &gt; zero; no overflow</td>
</tr>
<tr>
<td>c(GG R₁) = c(GG R₁) ± c(doubleword in memory)</td>
<td>3: Result has overflowed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC Setting and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 1,‘F’2147483647’ 2³¹-1</td>
<td>0: Result is zero; no overflow</td>
</tr>
<tr>
<td>A 1,‘F’1’</td>
<td>1: Result is &lt; zero; no overflow</td>
</tr>
<tr>
<td>L 2,‘F’2147483647’ 2³¹-1</td>
<td>2: Result is &gt; zero; no overflow</td>
</tr>
<tr>
<td>S 2,‘F’1’</td>
<td>3: Result has overflowed</td>
</tr>
<tr>
<td>L 3,‘F’-2147483648’ -2³¹</td>
<td>0: Result is zero; no overflow</td>
</tr>
<tr>
<td>S 3,‘F’1’</td>
<td>1: Result is &lt; zero; no overflow</td>
</tr>
<tr>
<td>L 4,‘X’700000000 00000000’</td>
<td>2: Result is &gt; zero; no overflow</td>
</tr>
<tr>
<td>AGR 4,4</td>
<td>3: Result has overflowed</td>
</tr>
<tr>
<td>LG 5,‘X’500000000 00000000’</td>
<td>0: Result is zero; no overflow</td>
</tr>
<tr>
<td>SG 5,‘X’600000000 00000000’</td>
<td>1: Result is &lt; zero; no overflow</td>
</tr>
</tbody>
</table>

---

**Notes**
Signed-Arithmetic Compare Instructions

- Comparisons with 32-bit operands
  - CH, CHY: Compare \( c(\text{GR } R_1) \) to halfword in memory
  - C, CY: Compare \( c(\text{GR } R_1) \) to word in memory
  - CR: Compare \( c(\text{GR } R_1) \) to word in \( c(\text{GR } R_2) \)

- Comparisons with 64-bit operands
  - CG: Compare \( c(\text{GG } R_1) \) to doubleword in memory
  - CGR: Compare \( c(\text{GG } R_1) \) to doubleword in \( c(\text{GG } R_2) \)

- Condition Code settings:
  - The CC cannot be set to 3 as a result of a compare instruction

Logical-Arithmetic Add and Subtract Instructions

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

- Instructions with 32- and 64-bit operands

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Add word from memory</td>
<td>SL</td>
<td>Subtract word from memory</td>
</tr>
<tr>
<td>ALR</td>
<td>Add word from ( c(\text{GR } R_2) )</td>
<td>SLR</td>
<td>Subtract word from ( c(\text{GR } R_2) )</td>
</tr>
<tr>
<td>ALG</td>
<td>Add doubleword from memory</td>
<td>SLG</td>
<td>Subtract doubleword from memory</td>
</tr>
<tr>
<td>ALGR</td>
<td>Add doubleword from ( c(\text{GG } R_2) )</td>
<td>SLGR</td>
<td>Subtract doubleword from ( c(\text{GG } R_2) )</td>
</tr>
</tbody>
</table>

- The CC settings:

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC Setting and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c(\text{GR } R_1) = c(\text{GR } R_1) \pm c(\text{GR } R_2) )</td>
<td>0: Zero result, no carry (Note)</td>
</tr>
<tr>
<td>( c(\text{GR } R_1) = c(\text{GR } R_1) \pm c(\text{word in memory}) )</td>
<td>1: Nonzero result, no carry</td>
</tr>
<tr>
<td>( c(\text{GG } R_1) = c(\text{GG } R_1) \pm c(\text{GG } R_2) )</td>
<td>2: Zero result, carry</td>
</tr>
<tr>
<td>( c(\text{GG } R_1) = c(\text{GG } R_1) \pm c(\text{doubleword in memory}) )</td>
<td>3: Nonzero result, carry</td>
</tr>
</tbody>
</table>

Note: CC0 cannot occur for logical subtraction

Notes
Add With Carry, Subtract With Borrow

- Instructions with 32- and 64-bit operands

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Add word from memory</td>
<td>SL</td>
<td>Subtract word from memory</td>
</tr>
<tr>
<td>ACLR</td>
<td>Add word from c(GR R2)</td>
<td>SLR</td>
<td>Subtract word from c(GR R2)</td>
</tr>
<tr>
<td>ALGR</td>
<td>Add doubleword from memory</td>
<td>SLGR</td>
<td>Subtract doubleword from memory</td>
</tr>
<tr>
<td>ALCGR</td>
<td>Add doubleword from c(GG R1)</td>
<td>SLBGR</td>
<td>Subtract doubleword from c(GG R1)</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>Load A in register pair</td>
</tr>
<tr>
<td>AL</td>
<td>Logically add low-order part of B</td>
</tr>
<tr>
<td>ALC</td>
<td>Add high-order part of B with carry</td>
</tr>
<tr>
<td>STM</td>
<td>Store the double-length sum</td>
</tr>
<tr>
<td>LM</td>
<td>Get first operand</td>
</tr>
<tr>
<td>SL</td>
<td>Logically subtract low-order parts</td>
</tr>
<tr>
<td>SLB</td>
<td>Subtract high-order parts with borrow</td>
</tr>
<tr>
<td>STM</td>
<td>Store 64-bit difference</td>
</tr>
</tbody>
</table>

- These instructions are especially useful for multi-precision arithmetic

Notes

Operations With Mixed 64-Bit and 32-Bit Operands

- 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_i

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Function</th>
<th>Mnem</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGF</td>
<td>Add word from memory</td>
<td>SGF</td>
<td>Subtract word from memory</td>
</tr>
<tr>
<td>AGFR</td>
<td>Add word from c(GR R2)</td>
<td>SGFR</td>
<td>Subtract word from c(GR R2)</td>
</tr>
<tr>
<td>CGF</td>
<td>Compare to word from memory</td>
<td>CGFR</td>
<td>Compare to word from c(GR R2)</td>
</tr>
<tr>
<td>ALGF</td>
<td>Logical add word from memory</td>
<td>SLGF</td>
<td>Logical subtract word from memory</td>
</tr>
<tr>
<td>ALGFR</td>
<td>Logical add word from c(GR R2)</td>
<td>SLGFR</td>
<td>Logical subtract word from c(GR R2)</td>
</tr>
</tbody>
</table>

- These instructions can simplify programs with mixed-length operands

Notes
Logical-Arithmetic Compare Instructions

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL</td>
<td>Logical compare c(GR $R_1$) to word from memory</td>
</tr>
<tr>
<td>CLR</td>
<td>Logical compare c(GR $R_1$) to word from c(GR $R_2$)</td>
</tr>
<tr>
<td>CLG</td>
<td>Logical compare c(GG $R_1$) to doubleword from memory</td>
</tr>
<tr>
<td>CLGR</td>
<td>Logical compare c(GG $R_1$) to doubleword from c(GG $R_2$)</td>
</tr>
<tr>
<td>CLGF</td>
<td>Logical compare c(GG $R_1$) to zero-extended word from memory</td>
</tr>
<tr>
<td>CLGFR</td>
<td>Logical compare c(GG $R_1$) to zero-extended word from c(GR $R_2$)</td>
</tr>
<tr>
<td>CLM, CLMY</td>
<td>Logical compare bytes from low half of c(GG $R_1$) to bytes in memory</td>
</tr>
<tr>
<td>CLMH</td>
<td>Logical compare bytes from high half of c(GG $R_1$) to bytes in memory</td>
</tr>
</tbody>
</table>

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

Notes

Retrieving and Setting the Program Mask

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

| IPM $R_1$ | Insert CC and Program Mask into GR $R_1$ |
| SPM $R_1$ | Set CC and Program Mask from GR $R_1$ |

- "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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Exception Condition Controlled</th>
<th>Int. Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>36 (F)</td>
<td>Fixed-point overflow</td>
<td>B</td>
</tr>
<tr>
<td>37 (D)</td>
<td>Decimal overflow</td>
<td>A</td>
</tr>
<tr>
<td>38 (U)</td>
<td>Hexadecimal floating-point underflow</td>
<td>D</td>
</tr>
<tr>
<td>39 (S)</td>
<td>Hexadecimal floating-point lost significance</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPM</th>
<th>Set CC and Program Mask bits to zero</th>
</tr>
</thead>
</table>

Notes
Binary Shifting

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Action</th>
<th>Mnem</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift left logical</td>
<td>SRL</td>
<td>Shift right logical</td>
</tr>
<tr>
<td>RLL</td>
<td>Rotate left logical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLA</td>
<td>Shift left arithmetic</td>
<td>SRA</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>SLDL</td>
<td>Shift left double logical</td>
<td>SRDL</td>
<td>Shift right double logical</td>
</tr>
<tr>
<td>SLDA</td>
<td>Shift left double arithmetic</td>
<td>SRDA</td>
<td>Shift right double arithmetic</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Action</th>
<th>Mnem</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLLG</td>
<td>Shift left logical</td>
<td>SRLG</td>
<td>Shift right logical</td>
</tr>
<tr>
<td>RLLG</td>
<td>Rotate left logical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLAG</td>
<td>Shift left arithmetic</td>
<td>SRAG</td>
<td>Shift right arithmetic</td>
</tr>
</tbody>
</table>

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

Notes

---

Unit Shifts

- Assume an n-bit register looks like this:

```
  0 1 2 3 N-4 N-3 N-2 N-1
A B C D W X Y Z
```

- A unit shift moves bits left or right by one position

- Logical shifts:

  - The Condition Code is unchanged

Notes
• 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

Notes

---

**Single-Length Logical Shifts**

• For SLL and SRL: operands are R_i, D_j(B)
  
  \[
  \begin{align*}
  \text{L 7,} & X'87654321' \\
  \text{SLL 7,} & 5(0) \\
  \text{L 6,} & X'87654321' \\
  \text{SRL 6,} & 5(0)
  \end{align*}
  \]
  
  Initial contents of GR7
  Results: c(GR7) = X'ECA86420'
  Initial contents of GR6
  Results: c(GR6) = X'043B2A19'

• For SLLG and SRLG: operands are R_i, R_j, D_j(B)
  
  - The operand in GG R_j is shifted, the result is placed in GG R_i
  
  \[
  \begin{align*}
  \text{LG 1,} & X'123456789ABCDEF0' \\
  \text{SLLG 0,} & 1,9 \\
  \text{LG 1,} & X'123456789ABCDEF0' \\
  \text{SRLG 0,} & 1,9
  \end{align*}
  \]
  
  c(GG1) initialized
  c(GG0) = X'68ACF135 79BDE000'
  c(GG1) initialized
  c(GG0) = X'00091AB 3C4SE6F'
  
  - Because the R_i and R_j operands differ, c(GG1) remains unchanged

• Exercise: Verify the example results

Notes

---
Double-Length Logical Shifts

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

```
  a b c ... l m n o p ... y z
```

- **LM 0,1,** `X'123456789ABCDEFO'** GR(0,1) initialized
- **SLDL 0,9** `c(GR0) = X'68ACF135', c(GR1) = X'79BDE000'**
- **LM 2,3,** `X'123456789ABCDEFO'** GR(2,3) initialized
- **SRDL 2,9** `c(GR0) = X'000091A28', 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 Shift Instructions

- Arithmetic shifts always set the CC:

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC Setting and Meaning</th>
</tr>
</thead>
</table>
| Left shift| 0: Result is zero  
3: Result has overflowed |
| Right shift| 0: Result is zero  
2: Result is > zero |

- **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 (overflow)`

- **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, R, D, (B)

```
L 0,=A('X'56789ABC')  Load initial data into GR0
RLL 1,0,10 Rotate 10 bits, result in GR1
− Then c(GR1) = 'X'2E6AF159'
LG 0,=AD('X'56789ABCDEF01234') Initialize GG0
RLLG 1,0,10 Rotate 10 bits, result in GG1
− Then c(GG1) = 'X'2E6AF37BC048D159'
```

- None of the rotating-shift instructions changes the CC

### Calculated Shift Amounts

- Some shift amounts must be determined at execution time
- Solution: put the shift amount in the B2 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 ≤ N < 31

```
L 1,N Get small integer N
L 0,='1' Put a 1-bit at right end of GR0 (2^1)
SLL 0,0(1) Leave 2^N in GR0
− Exercise: What will happen if N ≥ 31?
```
An extended form of length modifier lets you specify lengths in bits: put a period (.) after the modifier L

- DC FL3'8' and DC FL24'8' are equivalent
- 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 × Multiplier = Product
- 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:
  - Divisor ) Dividend = Quotient
  - 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
Arithmetic (Signed) Multiplication Instructions

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Multiply (32×32→32×32)</td>
<td>MR</td>
<td>Multiply Register (32×32→32×32)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
R_1 \text{ (EVEN)} & R_1+1 \text{ (ODD)} & \text{MULTIPLIER} \\
32 & 63 & 32 & 63 & R_2 \text{ or } D_2(X2,B2) \text{ (IN REGISTER OR MEMORY)}
\end{array}
\]

- Example(1): Square the number in GR1
  \[
  MR \ 0,1 \ c(GR0,GR1) = c(GR1)*c(GR1)
\]
- Example(2): Square the number in GR0
  \[
  LR \ 1,0 \ \text{Copy } c(GR0) \text{ to GR1 (odd register of pair)}
  \]

Notes

Single-Length Arithmetic Products

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>Multiply Halfword (32×16)</td>
<td>MSR</td>
<td>Multiply Single Register</td>
</tr>
<tr>
<td>MS,</td>
<td>Multiply Single (32×32)</td>
<td>MSR</td>
<td>(32×32)</td>
</tr>
<tr>
<td>MSY</td>
<td></td>
<td>MSG</td>
<td>Multiply Single Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSGR</td>
<td>(64×64)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSGF</td>
<td>Multiply Single Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSGFR</td>
<td>(64×64)</td>
</tr>
</tbody>
</table>

- Examples

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MH</td>
<td>5,&quot;'H100&quot;</td>
<td>c(GR5)*100</td>
</tr>
<tr>
<td>LH</td>
<td>1,A</td>
<td>LOAD C(N) 32 TO GR</td>
</tr>
<tr>
<td>MSR</td>
<td>1,1</td>
<td>c(GR1)*c(GR1) = N</td>
</tr>
<tr>
<td>LG</td>
<td>1,&quot;F'D23456789&quot;</td>
<td>c(GR1) = 23456789</td>
</tr>
<tr>
<td>MS</td>
<td>1,&quot;F'D23456789&quot;</td>
<td>23456789+32 Bits</td>
</tr>
<tr>
<td>MSFR</td>
<td>1,&quot;F'D23456789&quot;</td>
<td>c(GR1) = 23456789</td>
</tr>
</tbody>
</table>

Notes
Logical (Unsigned) Multiplication Instructions

- For 64-bit products of two 32-bit unsigned operands
  
<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>Multiply Logical (32+32×32)</td>
<td>MLR</td>
<td>Multiply Logical Register (32+32×32)</td>
</tr>
</tbody>
</table>

- For 128-bit products of two 64-bit unsigned operands
  
<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLG</td>
<td>Multiply Logical (64+64×64)</td>
<td>MLGR</td>
<td>Multiply Logical Register (64+64×64)</td>
</tr>
</tbody>
</table>

Examples:

- Logical multiplication: \((2^{**32}-1)\times(2^{**32}-1) = 18446744065119617025\)
  
  | L 1,=F-1′ c(GR1) = X′ FFFFFFFF′ |
  | MLR 0,1 c(GR0,GR1) = X′ FFFFFFFF′ 00000001′ |
  | LG 1,=FD′74296604373′ c(GG1) = 74296604373 |
  | MLG 0,=FD′9876543210′ c(GG0,GG1) = 733793623446209457330 |

Notes

Division Instructions

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

  \[987^*867 = 855729; 855729/123 = 6957, \text{remainder 18}\]

- 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:

  \[
  \begin{array}{c|c}
  R_1 & R_1+1 \\
  \hline
  \text{Remainder} & \text{Quotient} \\
  \end{array}
  \]

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

```
<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Divide (32,32,32,32)</td>
<td>DR</td>
<td>Divide Register (32,32,32,32)</td>
</tr>
</tbody>
</table>
```

- The results are in an even-odd register pair

```
<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Divide (32,32,32,32)</td>
<td>DR</td>
<td>Divide Register (32,32,32,32)</td>
</tr>
</tbody>
</table>
```

Example:

```assembly
L 2,=F'-'14352'  Dividend in GR2
SRDA 2,32  Create double-length dividend
```

Notes
Logical (Unsigned) Division Instructions

- These instructions consider all operands unsigned:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL</td>
<td>Divide Logical (32,32 ÷ 32,32)</td>
<td>DLR</td>
<td>Divide Logical Register (32,32 ÷ 32,32)</td>
</tr>
<tr>
<td>DLG</td>
<td>Divide Logical (64,64 ÷ 64,64)</td>
<td>DLGR</td>
<td>Divide Logical Register (64,64 ÷ 64,64)</td>
</tr>
</tbody>
</table>

- The dividend is always double-length
- Example:
  
  ```
  L 0,=F'−2'
  SR 1,1
  DL 0,=F'−1'
  
  Set GR0 to X'FFFFFFFF'
  Set GR1 to X'00000000'
  Divide logically by X'FFFFFFFF'
  * Quotient and remainder = X'FFFFFFFF'
  ```

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

Summary of Multiply and Divide Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Product length (bits)</th>
<th>32</th>
<th>32 ÷ 32</th>
<th>64</th>
<th>64 ÷ 64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operand 1 length</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Operand 2 length</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Arithmetic ×</td>
<td>Mi</td>
<td>Ms</td>
<td>M</td>
<td>MsF</td>
<td>MsG</td>
</tr>
<tr>
<td>Logical ×</td>
<td>ML</td>
<td>MLR</td>
<td>MLG</td>
<td>MLGR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Dividend length (bits)</th>
<th>32 ÷ 32</th>
<th>64</th>
<th>64 ÷ 64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divisor length</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Quotient &amp; remainder length</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Arithmetic +</td>
<td>D</td>
<td>DR</td>
<td>DSF</td>
<td>DSFR</td>
</tr>
<tr>
<td>Logical +</td>
<td>DL</td>
<td>DLR</td>
<td>DLSF</td>
<td>DLSFR</td>
</tr>
</tbody>
</table>

Notes
Logical Operations

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

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XOR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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

Register-Based Logical Instructions

- Instructions with 32-bit operands:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, NY</td>
<td>AND (32)</td>
<td>N/R</td>
<td>AND Register (32)</td>
</tr>
<tr>
<td>O, OY</td>
<td>OR (32)</td>
<td>OR</td>
<td>OR Register (32)</td>
</tr>
<tr>
<td>X, XY</td>
<td>Exclusive OR (32)</td>
<td>XR</td>
<td>Exclusive OR Register (32)</td>
</tr>
</tbody>
</table>

- Instructions with 64-bit operands:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NG</td>
<td>AND (64)</td>
<td>NGR</td>
<td>AND Register (64)</td>
</tr>
<tr>
<td>DG</td>
<td>OR (64)</td>
<td>DGR</td>
<td>OR Register (64)</td>
</tr>
<tr>
<td>XG</td>
<td>Exclusive OR (64)</td>
<td>XGR</td>
<td>Exclusive OR Register (64)</td>
</tr>
</tbody>
</table>

- Each instruction sets the Condition Code:

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND OR XOR</td>
<td>0: all result bits are zero 1: result bits are not all zero</td>
</tr>
</tbody>
</table>

Notes
Examples of AND, OR, and XOR

- Consider each operation, using identical operands:

<table>
<thead>
<tr>
<th>Operation</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>NR 4,9</td>
<td>OR 4,9</td>
<td>XR 4,9</td>
</tr>
<tr>
<td>c(GR4)</td>
<td>X’01234567’</td>
<td>X’01234567’</td>
<td>X’01234567’</td>
</tr>
<tr>
<td>c(GR9)</td>
<td>X’EDA96521’</td>
<td>X’EDA96521’</td>
<td>X’EDA96521’</td>
</tr>
<tr>
<td>Result</td>
<td>X’01214521’</td>
<td>X’EDA96521’</td>
<td>X’EC8A2046’</td>
</tr>
</tbody>
</table>

- To see in more detail how these results are obtained, examine the fourth hexadecimal digit (3 and 9) for each case:

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 0011</td>
<td>3 0011</td>
<td>3 0011</td>
</tr>
<tr>
<td>9 1001</td>
<td>9 1001</td>
<td>9 1001</td>
</tr>
<tr>
<td>1 0001</td>
<td>8 1011</td>
<td>A 1010</td>
</tr>
</tbody>
</table>

Notes

Interesting Uses of Logical Instructions

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’

Notes
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.

Section 20: Address Generation and Addressing Modes

- 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:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>DL2</th>
<th>DH2</th>
<th>opcode</th>
</tr>
</thead>
</table>

• Address generation first creates a signed displacement:

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>X</th>
<th>B</th>
<th>DL</th>
<th>S</th>
<th>DH</th>
</tr>
</thead>
</table>

SIGN-EXTENDED | S | DH | DL |

64-BIT SIGNED DISPLACEMENT

ADD TO C(BASE REGISTER B)

EFFECTIVE ADDRESS

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

Notes

---

Address Generation: Relative-Immediate Operands

• Relative-immediate instructions have two basic formats:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>OP</th>
<th>RI2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>OP</th>
<th>RI2</th>
</tr>
</thead>
</table>

| OPCODE, REGS | SBBBBBBBBBBBBB |

RI-TYPE INSTRUCTION

SHIFT LEFT 1 BIT

SIGN-EXTENDED | SBBBBBBBBBBBB |

64-BIT SIGNED OFFSET

ADD TO ADDRESS OF THE INSTRUCTION ITSELF (NOT THE PSW'S IA!)

EFFECTIVE ADDRESS

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

Notes

---
Addressing Modes

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

<table>
<thead>
<tr>
<th>AMODE</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>24-BIT ADDRESS</td>
</tr>
<tr>
<td>31</td>
<td>31-BIT ADDRESS</td>
</tr>
<tr>
<td>64</td>
<td>64-BIT ADDRESS</td>
</tr>
</tbody>
</table>

Notes

Load Address Instructions

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA, LAY</td>
<td>Load Address</td>
<td>LARL</td>
<td>Load Address Relative Long</td>
</tr>
</tbody>
</table>

- Small constants can be put in a GR using LA, LAY:
  - $LA 0, n(0,0)$ $0 \leq n \leq 4095$
  - $LAY 0, n(0,0)$ $-2^{19} \leq n \leq 2^{19}-1$
- These instructions are modal: the results depend on the AMode
  - 24-bit mode: result in GR0 is 'X'00FFFFFF'
  - 31-bit mode: result in GR0 is 'X'7FFFFFFFF'
  - 64-bit mode: result in GG0 is 'X'FFFFFFFFFFFFFF'
- LARL always creates a memory address (relative to LARL)

Notes
Summary

This table summarizes the three Load Address instructions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Instruction</th>
<th>Result in R, general register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Address</td>
<td>LA</td>
<td>Effective Address in bits 40-63; zero in bits 32-39; bits 0-31 unchanged.</td>
</tr>
<tr>
<td>based (based)</td>
<td>LAY</td>
<td>Effective Address in bits 33-63; zero in bit 32; bits 0-31 unchanged.</td>
</tr>
<tr>
<td>Load Address</td>
<td>LARL</td>
<td>Effective Address in bits 0-63.</td>
</tr>
<tr>
<td>relative (relative)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes

Immediate Operands

“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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>R1</th>
<th>OP</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RI</td>
<td>OPCODE</td>
<td>R1</td>
<td>OP</td>
</tr>
<tr>
<td>RIL</td>
<td>OPCODE</td>
<td>R1</td>
<td>OP</td>
</tr>
</tbody>
</table>

Some instructions affect an entire register, some only parts:

- H = High Half; HL = High Half’s Low Half, etc.
Logical-Immediate Insert Instructions

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>HH</td>
<td>Insert Logical Immediate (high) (64→32)</td>
<td>LF</td>
<td>Insert Logical Immediate (low) (64→32)</td>
</tr>
<tr>
<td>HH</td>
<td>Insert Logical Immediate (high) (64→16)</td>
<td>HL</td>
<td>Insert Logical Immediate (high low) (64→16)</td>
</tr>
<tr>
<td>LL</td>
<td>Insert Logical Immediate (low high) (64→16)</td>
<td>LF</td>
<td>Insert Logical Immediate (low low) (64→16)</td>
</tr>
</tbody>
</table>

Notes

Arithmetic- and Logical-Immediate Load Instructions

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHI</td>
<td>Load Halfword Immediate (32→16)</td>
<td>LGHI</td>
<td>Load Halfword Immediate (64→16)</td>
</tr>
<tr>
<td>LGFI</td>
<td>Load Immediate (64→32)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLIF</td>
<td>Load Logical Immediate (low) (64→32)</td>
<td>LLHL</td>
<td>Load Logical Immediate (low high) (64→16)</td>
</tr>
<tr>
<td>LLIF</td>
<td>Load Logical Immediate (high) (64→32)</td>
<td>LLHL</td>
<td>Load Logical Immediate (low low) (64→16)</td>
</tr>
<tr>
<td>LLIF</td>
<td>Load Logical Immediate (high high) (64→16)</td>
<td>LLHL</td>
<td>Load Logical Immediate (low low) (64→16)</td>
</tr>
</tbody>
</table>

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

Notes
## Arithmetic Instructions with Immediate Operands

- **Arithmetic and logical add and subtract instructions**:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHI</td>
<td>Add Halfword Immediate (32←16)</td>
</tr>
<tr>
<td>AGHI</td>
<td>Add Halfword Immediate (64←16)</td>
</tr>
<tr>
<td>AFI</td>
<td>Add Immediate (32)</td>
</tr>
<tr>
<td>AGFI</td>
<td>Add Immediate (64←32)</td>
</tr>
<tr>
<td>ALFI</td>
<td>Add Logical Immediate (32)</td>
</tr>
<tr>
<td>ALGFI</td>
<td>Add Logical Immediate (64←32)</td>
</tr>
<tr>
<td>SLFI</td>
<td>Subtract Logical Immediate (32)</td>
</tr>
<tr>
<td>SLGFI</td>
<td>Subtract Logical Immediate (64←32)</td>
</tr>
</tbody>
</table>

- **Arithmetic and logical compare instructions**:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHI</td>
<td>Compare Halfword Immediate (32←16)</td>
</tr>
<tr>
<td>CGHI</td>
<td>Compare Halfword Immediate (64←16)</td>
</tr>
<tr>
<td>CFI</td>
<td>Compare Immediate (32)</td>
</tr>
<tr>
<td>CGFI</td>
<td>Compare Immediate (64←32)</td>
</tr>
<tr>
<td>CLFI</td>
<td>Compare Logical Immediate (32)</td>
</tr>
<tr>
<td>CLGFI</td>
<td>Compare Logical Immediate (64←32)</td>
</tr>
</tbody>
</table>

- **Multiply instructions with an immediate operand**:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHI</td>
<td>Multiply Halfword Immediate (32←16)</td>
</tr>
<tr>
<td>MGHI</td>
<td>Multiply Halfword Immediate (64←16)</td>
</tr>
</tbody>
</table>

---

## Logical Operations with Immediate Operands

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIHF</td>
<td>AND Immediate (high) (64←32)</td>
</tr>
<tr>
<td>NILF</td>
<td>AND Immediate (low) (64←32)</td>
</tr>
<tr>
<td>NIHH</td>
<td>AND Immediate (high high) (64←16)</td>
</tr>
<tr>
<td>NIHL</td>
<td>AND Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>NIIH</td>
<td>AND Immediate (low high) (64←16)</td>
</tr>
<tr>
<td>NILL</td>
<td>AND Immediate (low low) (64←16)</td>
</tr>
<tr>
<td>OIHF</td>
<td>OR Immediate (high) (64←32)</td>
</tr>
<tr>
<td>OIFL</td>
<td>OR Immediate (low) (64←32)</td>
</tr>
<tr>
<td>OIHH</td>
<td>OR Immediate (high high) (64←16)</td>
</tr>
<tr>
<td>OIHL</td>
<td>OR Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>OILH</td>
<td>OR Immediate (low high) (64←16)</td>
</tr>
<tr>
<td>OILL</td>
<td>OR Immediate (low low) (64←16)</td>
</tr>
<tr>
<td>XIHF</td>
<td>XOR Immediate (high) (64←32)</td>
</tr>
<tr>
<td>XILF</td>
<td>XOR Immediate (low) (64←32)</td>
</tr>
</tbody>
</table>

---

Notes

---
Summary

- 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 bits</td>
<td>32 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>Arithmetic Add/Subtract</td>
<td>AHI</td>
<td>AF1</td>
<td>AGHI</td>
</tr>
<tr>
<td>Logical Add/Subtract</td>
<td>ALFI</td>
<td>SLFI</td>
<td>ALGFI</td>
</tr>
<tr>
<td>Arithmetic Compare</td>
<td>CHI</td>
<td>CF1</td>
<td>CGHI</td>
</tr>
<tr>
<td>Logical Compare</td>
<td>CLFI</td>
<td></td>
<td>CLGFI</td>
</tr>
<tr>
<td>Multiply</td>
<td>MHI</td>
<td></td>
<td>MGHI</td>
</tr>
</tbody>
</table>

Branches, Loops, and Indexing

- 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

Notes
• The BRC and BRCL instructions have these formats:

\[
\begin{array}{c|c|c|c}
A7 & M1 & 4 & RI2 \\
\hline
\end{array}
\]

- The branch target can be as far away as \(-65536\) and \(+65534\) bytes

\[
\begin{array}{c|c|c|c}
C0 & M1 & 4 & RI2 \\
\hline
\end{array}
\]

- 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.

---

### Relative Branch Extended Mnemonics

• 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:

<table>
<thead>
<tr>
<th>RI Mnemonic</th>
<th>RIL Mnemonic</th>
<th>Mask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRC</td>
<td>JC</td>
<td>M1</td>
<td>Conditional Branch</td>
</tr>
<tr>
<td>BRU</td>
<td>J</td>
<td></td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BRNO</td>
<td>JNO</td>
<td>14</td>
<td>Branch if No Overflow</td>
</tr>
<tr>
<td>BRNH</td>
<td>JNH</td>
<td>13</td>
<td>Branch if Not High</td>
</tr>
<tr>
<td>BRNP</td>
<td>JNP</td>
<td>13</td>
<td>Branch if Not Plus</td>
</tr>
<tr>
<td>BRNL</td>
<td>JNL</td>
<td>11</td>
<td>Branch if Not Low</td>
</tr>
<tr>
<td>BRNM</td>
<td>JNM</td>
<td>11</td>
<td>Branch if Not Minus</td>
</tr>
<tr>
<td>BRE</td>
<td>JE</td>
<td>8</td>
<td>Branch if Equal</td>
</tr>
<tr>
<td>BRP</td>
<td>JP</td>
<td>2</td>
<td>Branch if Plus</td>
</tr>
<tr>
<td>BRO</td>
<td>JO</td>
<td>1</td>
<td>Branch if Overflow</td>
</tr>
</tbody>
</table>

---
• 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:
    
    ```asm
    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

Notes

Branch on Count Instructions

• Count-controlled loops are easily managed with these instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCT</td>
<td>Branch on Count (32)</td>
<td>BCTR</td>
<td>Branch on Count Register (32)</td>
</tr>
<tr>
<td>BCTG</td>
<td>Branch on Count (64)</td>
<td>BCTGR</td>
<td>Branch on Count Register (64)</td>
</tr>
<tr>
<td>BRCT, JCT</td>
<td>Branch Relative on Count (32)</td>
<td>BRCTG, JCTG</td>
<td>Branch Relative on Count (64)</td>
</tr>
</tbody>
</table>

• Execution follows these steps:
  1. Reduce the number in the R1 register by one
     - For BCTR and BCTGR: if the R2 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)

  ```asm
  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
  ```

Notes
Looping in General

- There are many types of loop; some of the most common are:

  1. "Do-Until"

     ![Do-Until Diagram]

     - Initialize Index, Increment, Comparand
     - Loop Body
     - Add Increment to Index
     - Compare to Comparand
     - Done if Comparand NOT
       Equal to Index

  2. "Do-While"

     ![Do-While Diagram]

     - Initialize Index, Increment, Comparand
     - Loop Body
     - Add Increment to Index
     - Compare to Comparand
     - Not Done if
       Comparand NOT
       Equal to Index

  3. A combination

     ![Combination Diagram]

     - Initialize
     - Loop Body, First Part
     - Exit Test
     - Loop Body, Remainder

Notes
Examples Using BXLE

For all the “Branch on Index” instructions:
- The R1 operand is the index; the increment is in R3, and the comparand is in R3|1 (R3 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(j) 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
     ```

Notes

Examples Using BXH

- 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(j) 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
     ```

Notes
Specialized Uses of BXLE and BXH

- 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

Notes

Summary

- These instructions are described in Section 22:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relative-Immediate Operand Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Relative-Immediate Operand Length</td>
</tr>
<tr>
<td></td>
<td>16 bits</td>
</tr>
<tr>
<td>Branch on Condition (Relative)</td>
<td>BCR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bits</td>
</tr>
<tr>
<td>Branch on Count (Register)</td>
<td>BCTR</td>
</tr>
<tr>
<td>Branch on Count (Indexed)</td>
<td>BCT</td>
</tr>
<tr>
<td>Branch on Count (Relative)</td>
<td>BRCT</td>
</tr>
<tr>
<td>Branch on Index</td>
<td>BXH</td>
</tr>
<tr>
<td>Branch on Index (Relative)</td>
<td>BRXH</td>
</tr>
</tbody>
</table>

Notes
Bit and Character Data

- 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

Bit and Byte Data and Instructions

- 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
SI- and SIY-Type Instructions

- We'll discuss these instructions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonic</th>
<th>Action</th>
<th>CC set?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MVI, MVIY</td>
<td>Operand 1 ← I₂</td>
<td>No</td>
</tr>
<tr>
<td>AND</td>
<td>NI, NIY</td>
<td>Operand 1 ← Operand 1 AND I₂</td>
<td>Yes</td>
</tr>
<tr>
<td>OR</td>
<td>OI, OIY</td>
<td>Operand 1 ← Operand 1 OR I₂</td>
<td>Yes</td>
</tr>
<tr>
<td>XOR</td>
<td>XI, XIX</td>
<td>Operand 1 ← Operand 1 XOR I₂</td>
<td>Yes</td>
</tr>
<tr>
<td>Compare</td>
<td>CLI, CLIY</td>
<td>Operand 1 Compared to I₂</td>
<td>Yes</td>
</tr>
<tr>
<td>Test Under Mask</td>
<td>TM, TMY</td>
<td>Test Selected Bits of Operand 1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>I₂</th>
<th>B₁</th>
<th>D₁</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>I₂</th>
<th>B₁</th>
<th>DL</th>
<th>DH</th>
<th>opcode</th>
</tr>
</thead>
</table>

- Write the operand field as either D₁(B₁)I₂ or S₁I₂

MVI Instructions

- There are two Move Immediate instructions:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI</td>
<td>Move Immediate</td>
<td>MVIY</td>
<td>Move Immediate</td>
</tr>
</tbody>
</table>

- Each stores its I₂ 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' Printer carriage control for new page
NI, OI, and XI Instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NI</td>
<td>AND Immediate</td>
<td>NIY</td>
<td>AND Immediate</td>
</tr>
<tr>
<td>OI</td>
<td>OR Immediate</td>
<td>OIY</td>
<td>OR Immediate</td>
</tr>
<tr>
<td>XI</td>
<td>XOR Immediate</td>
<td>XIY</td>
<td>XOR Immediate</td>
</tr>
</tbody>
</table>

- Each instruction sets the Condition Code:

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0: all result bits are zero</td>
</tr>
<tr>
<td>OR</td>
<td>1: result bits are not all zero</td>
</tr>
<tr>
<td>XOR</td>
<td></td>
</tr>
</tbody>
</table>

(a) NI X,0 Same as ‘MVI X,0’ except CC set to 0
NI X,B’111111101’ 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

Notes

---

CLI Instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI</td>
<td>Compare Immediate</td>
<td>CLIY</td>
<td>Compare Immediate</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 = I</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 &lt; I</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 &gt; I</td>
</tr>
</tbody>
</table>

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

CLI =’C’’,X’CI’ CC = 0: c(Operand 1) = I,
CLI =’00’’,0 CC = 0: c(Operand 1) = I,
CLI =’C’’,B’01000000’ CC = 0: c(Operand 1) = I,
CLI =’X’’,X’2’ CC = 1: c(Operand 1) < I,
CLI =’C’’,X’250 CC = 1: c(Operand 1) < I,
CLI =’C’’,C’X’-1 CC = 2: c(Operand 1) > I,
CLI =’X’’,X’0’ CC = 2: c(Operand 1) > I

Notes

---
Test Under Mask Instructions

- The 1-bits of the immediate \( I_1 \) operand indicate which corresponding bits of the first operand will be tested; the CC shows the result.

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bits examined are all zero, or mask is zero</td>
</tr>
<tr>
<td>1</td>
<td>Bits examined are mixed zero and one</td>
</tr>
<tr>
<td>3</td>
<td>Bits examined are all one</td>
</tr>
</tbody>
</table>

| TM  | Num, \('X80'\)          | Test leftmost bit of a number |
| J0  | 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 |

Notes

• It's better to name bit-data items than to use bit numbers or bit masks.

• Discouraged techniques:
  
  or

• Better technique: name each flag bit separately

<table>
<thead>
<tr>
<th>Retired</th>
<th>Equ X'40'</th>
<th>Retired status flag bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FullTime</td>
<td>Equ X'20'</td>
<td>Full time worker status flag bit</td>
</tr>
<tr>
<td>PartTime</td>
<td>Equ X'10'</td>
<td>Part time worker status flag bit</td>
</tr>
<tr>
<td>Exempt</td>
<td>Equ X'08'</td>
<td>Exempt employee status flag bit</td>
</tr>
<tr>
<td>Hourly</td>
<td>Equ X'04'</td>
<td>Hourly employee status flag bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>

| Retired | Equ X'40' | Person has retired [Better Method] |

Notes
Avoiding Bit-Naming Problems

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

```assembly
Flag1 DS X │ Flag2 DS X
Bit0 Equ X'80' │ Bit1 Equ X'40'
```
- Normally we would write something like

```assembly
OI Flag1,Bit0 │ OI Flag2,Bit1
```
- But we could accidentally write (without assembler error!)

```assembly
OI Flag2,Bit0 │ OI Flag1,Bit1
```
- One way to associate specific bits with their “owning” bytes:

```assembly
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

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

Notes

Instruction Modification

- You may see (or be tempted to write) self-modifying programs

1. Skip some statements after they’re executed once

```assembly
NOP NOP SkipIt │ Fall through first time
OI NOP+1,X'F0' │ Change NOP to unconditional branch
    SkipIt DC OH │ Continue execution
```
- Alternating between branching or not

```assembly
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

Notes
### Summary

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12-bit displacement</td>
<td>20-bit displacement</td>
</tr>
<tr>
<td>Move Immediate</td>
<td>MVI</td>
<td>MVIY</td>
</tr>
<tr>
<td>AND Immediate</td>
<td>NI</td>
<td>NIY</td>
</tr>
<tr>
<td>OR Immediate</td>
<td>OI</td>
<td>OIY</td>
</tr>
<tr>
<td>XOR Immediate</td>
<td>XI</td>
<td>XIY</td>
</tr>
<tr>
<td>Compare Immediate</td>
<td>CLI</td>
<td>CLIY</td>
</tr>
<tr>
<td>Test Under Mask</td>
<td>TM</td>
<td>TMY</td>
</tr>
</tbody>
</table>

### Notes

**Character Data and Basic Instructions**

- 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
Basic SS-Type Instructions

- Start with these typical SS-type instructions:

<table>
<thead>
<tr>
<th>Mnemon</th>
<th>Instruction</th>
<th>Mnemon</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC</td>
<td>Move [Characters]</td>
<td>MVCIN</td>
<td>Move [Characters] Inverse</td>
</tr>
<tr>
<td>NC</td>
<td>AND [Characters]</td>
<td>OC</td>
<td>OR [Characters]</td>
</tr>
<tr>
<td>XC</td>
<td>XOR [Characters]</td>
<td>CLC</td>
<td>Compare Logical [Characters]</td>
</tr>
<tr>
<td>TR</td>
<td>Translate</td>
<td>TRT</td>
<td>Translate and Test</td>
</tr>
<tr>
<td>TRTR</td>
<td>Translate and Test Reverse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each has this machine-instruction format:

```
opcode    L    B1   D1    B2   D2
```

- The basic assembler instruction statement format is:

```
mnemonic D1(B1),D2(B2)
```

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

Notes

Operand Specifications and Explicit Lengths

- 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
S1(N),S2
D1(N,B1),S2
S1(N),D2(B2)
D1(N,B1),D2(B2)
```

- Examples:

```
MVC     B(B23),AA     S1(N),S2
MVC     X'478'(23,9),AA D1(N,B1),S2
MVC     B(B23),X'125'(9) S1(N),D1(B1)
MVC     1149(23,9),293(9) D1(N,B1),D2(B2)
```

Notes
Symbol Length Attribute References

- Written as $L'$ followed by a symbol

$$\text{LA } 0, L' \text{BB } C(\text{GR}) = \text{Length Attribute of BB}$$

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

   \[ \begin{array}{ll}
   \text{G} & \text{Equ } * \\
   \text{H} & \text{Equ } *,20
   \end{array} \]

   Length attribute of G = 1
   Length attribute of H = 20

2. Literals have the length attribute of the first operand

   $$\text{LA } 0, \text{'X'123456,ABC,FEDCBA' } c(\text{GR0}) = 3$$

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

   $$\text{MVC BB(L' *),AA Length attribute of MVC = 6}$$

Notes

Implied Lengths

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

<table>
<thead>
<tr>
<th>Implied Length</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1S_2$</td>
<td></td>
</tr>
<tr>
<td>$D_1(B_1)S_2$</td>
<td></td>
</tr>
<tr>
<td>$S_1D_2(B_1)$</td>
<td></td>
</tr>
<tr>
<td>$D_1(B_1)D_2(B_1)$</td>
<td></td>
</tr>
</tbody>
</table>

- Implied lengths simplify specifying how many bytes are involved

  $$\text{MVC AA, BB Move L'AA bytes from BB to AA}$$

- Summary of explicit/implied addresses and lengths

<table>
<thead>
<tr>
<th>First operand form</th>
<th>Address specification</th>
<th>Length Expression</th>
<th>Length used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>implied</td>
<td>implied</td>
<td>$L'S_1$</td>
</tr>
<tr>
<td>$S_1(N)$</td>
<td>implied</td>
<td>explicit</td>
<td>N</td>
</tr>
<tr>
<td>$D_1(B_1)$</td>
<td>explicit</td>
<td>implied</td>
<td>$L'D_1$</td>
</tr>
<tr>
<td>$D_1(N,B_1)$</td>
<td>explicit</td>
<td>explicit</td>
<td>N</td>
</tr>
</tbody>
</table>

Notes
The Encoded Length “L” and Program Length “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

Notes

---

The MVC and MVCIN Instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVC</td>
<td>Move [Characters]</td>
<td>MVCIN</td>
<td>Move [Characters] Inverse</td>
</tr>
</tbody>
</table>

- These instructions move 1-256 bytes (0 ≤ **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

Notes
The NC, OC, and XC Instructions

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND OR XOR</td>
<td>0: all result bits are zero 1: result bits are not all zero</td>
</tr>
</tbody>
</table>

• 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

Notes

The CLC Instruction

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

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 = Operand 2</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 &lt; Operand 2</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 &gt; Operand 2</td>
</tr>
</tbody>
</table>

• Example: If the 120 bytes at Line contain blanks, branch to AllBlank

  CLC LINE(120),=CL120' COMPARE TO 120 BLANKS OR
  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)

Notes
The TR (Translate) Instruction

- 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, 'C0123456789,6C' ' Translate table
Data DC C' A1Gp374+SN6/7' Result = C' 1 2 3 4 5 6 7'
```

Notes

The TRT and TRTR Instructions

- 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:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All accessed function bytes were zero.</td>
</tr>
<tr>
<td>1</td>
<td>A nonzero function byte was accessed before the last argument byte was reached.</td>
</tr>
<tr>
<td>2</td>
<td>The nonzero function byte accessed corresponds to the last argument byte.</td>
</tr>
</tbody>
</table>

- Example: scan the table at Data for numeric characters

```
TRT Data,NumTable Scan Data for numeric characters
```

```
NumTable DC 240X, '0',10X1',6X0' ' Detect numeric characters
Data DC C' A1Gp374+SN6/7' Data to be scanned for numerics
```

Notes
The Execute Instructions (1)  

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>Execute</td>
<td>EXRL</td>
<td>Execute Relative Long</td>
</tr>
</tbody>
</table>

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

Notes

The Execute Instructions (2)

- 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 ($N\rightarrow N-1$)
  - 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 $M_1$
  - 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

Notes
Summary

- Instructions discussed in Section 24:

<table>
<thead>
<tr>
<th>Function</th>
<th>Instruction</th>
<th>Data is Processed</th>
<th>CC Set?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MVC</td>
<td>Left to right</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>MVCIN</td>
<td>Right to left</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>NC</td>
<td>Left to right</td>
<td>Yes</td>
</tr>
<tr>
<td>OR</td>
<td>OC</td>
<td>Left to right</td>
<td>Yes</td>
</tr>
<tr>
<td>XOR</td>
<td>OC</td>
<td>Left to right</td>
<td>Yes</td>
</tr>
<tr>
<td>Compare</td>
<td>CLC</td>
<td>Left to right</td>
<td>Yes</td>
</tr>
<tr>
<td>Translate</td>
<td>TR</td>
<td>Left to right</td>
<td>No</td>
</tr>
<tr>
<td>Translate and Test</td>
<td>TRT</td>
<td>Left to right</td>
<td>Yes</td>
</tr>
<tr>
<td>Translate and Test Reverse</td>
<td>TRTR</td>
<td>Right to left</td>
<td>Yes</td>
</tr>
<tr>
<td>Execute</td>
<td>EX</td>
<td>—</td>
<td>Depends on target</td>
</tr>
<tr>
<td></td>
<td>EXRL</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Notes

Character Data and Extended Instructions

- 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

Notes
Move Long and Compare Logical Long

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVCL</td>
<td>Move Long</td>
<td>CLCL</td>
<td>Compare Logical Long</td>
</tr>
</tbody>
</table>

MVCL R₁, R₂  and  CLCL R₁, R₂

- 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 R₂+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 destructive overlap is possible:
    - part of the target field is used as source data after data has been moved into it

Notes

---

The MVCL Instruction

- 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(R₁+1) is 0 before the source length c(R₂+1), set CC=1
  4. If the source length c(R₂+1) is 0 before the target length c(R₁+1), use the pad character as source data until the target length is 0; set CC=2

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 length = Operand 2 length</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 length = Operand 2 length; part of Operand 2 not moved</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 length = Operand 2 length; Operand 1 was padded</td>
</tr>
<tr>
<td>3</td>
<td>Destructive Overlap, no data movement</td>
</tr>
</tbody>
</table>

- Example: Set 2400 bytes at Field to zeros
  
  | LA | 0, Field c(R₁) = Target address |
  | LNI | 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

Notes
The CLCL Instruction

- 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

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 = Operand 2, or both lengths 0</td>
</tr>
<tr>
<td>1</td>
<td>First Operand low</td>
</tr>
<tr>
<td>2</td>
<td>First Operand high</td>
</tr>
</tbody>
</table>

- 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
```

Notes

---

Move Long and Compare Logical Long Extended

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

  - The \( R_1 \) and \( R_1 \) 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,R1,0,D(8) Target,source,pad_character
  ```

  as in

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

Notes

---
The MVCLE Instruction

- 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

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 length = Operand 2 length</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 length &lt; Operand 2 length; part of operand 2 not moved</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 length &gt; Operand 2 length; operand 1 was padded</td>
</tr>
<tr>
<td>3</td>
<td>CPU wants to rest; branch back to the MVCLE</td>
</tr>
</tbody>
</table>

- Example: set 2400 bytes at Field to zeros

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LA 0,Field</td>
<td>c(R) = Target address</td>
</tr>
<tr>
<td>LHI 1,2400</td>
<td>c(R_1)+1 = Target length</td>
</tr>
<tr>
<td>SR 3,3</td>
<td>c(R_3)+1 = Source length = 0</td>
</tr>
<tr>
<td>SR 5,5</td>
<td>c(R5) = Pad byte = X’00’</td>
</tr>
</tbody>
</table>

* No source address is required if source length is zero

MVCLE 0,2,0(5) Move pad bytes to target Field

JE Cleared Branch if the Field was all zeros

Notes

The CLCLE Instruction

- 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

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 = Operand 2, or both 0 length</td>
</tr>
<tr>
<td>1</td>
<td>First operand low</td>
</tr>
<tr>
<td>2</td>
<td>First operand high</td>
</tr>
<tr>
<td>3</td>
<td>No inequality found thus far; operands are not exhausted</td>
</tr>
</tbody>
</table>

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LA 0,Field</td>
<td>c(R) = Target address</td>
</tr>
<tr>
<td>LHI 1,2400</td>
<td>c(R_1)+1 = Target length</td>
</tr>
<tr>
<td>SR 3,3</td>
<td>c(R_3)+1 = Source length = 0</td>
</tr>
<tr>
<td>SR 5,5</td>
<td>Set pad character to X’00’</td>
</tr>
</tbody>
</table>

* 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

Notes
Special “C-String” Instructions

- 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 ’A C-string.’,X’0’ Generates ’A C-string.

- These four instructions simplify working with C-strings:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVST</td>
<td>Move String</td>
<td>CLST</td>
<td>Compare Logical String</td>
</tr>
<tr>
<td>SRST</td>
<td>Search String</td>
<td>TRE</td>
<td>Translate Extended</td>
</tr>
</tbody>
</table>

- 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

Notes

Search String Instruction

- 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 R2
  3. One byte past the end of the string is placed in R1 (to limit the search)
- Condition Code settings:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test character found; R1 points to it</td>
</tr>
<tr>
<td>2</td>
<td>Test character not found before the byte addressed by R1</td>
</tr>
<tr>
<td>3</td>
<td>Partial search with no match; R1 unchanged, R2 points to next byte to process</td>
</tr>
</tbody>
</table>

- 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,’(’ 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

Notes
Move String Instruction

- 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 R1
  3. The source-string address is placed in R2
- The Condition Code settings are:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Entire second operand moved; R1 points to end of first operand</td>
</tr>
<tr>
<td>3</td>
<td>Incomplete move; R1 and R2 point to next bytes to process</td>
</tr>
</tbody>
</table>

- 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

Compare Logical String Instruction

- 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:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Entire operands are equal; R1 and R2 unchanged</td>
</tr>
<tr>
<td>1</td>
<td>First operand low; R1 and R2 point to last bytes processed</td>
</tr>
<tr>
<td>2</td>
<td>First operand high; R1 and R2 point to last bytes processed</td>
</tr>
<tr>
<td>3</td>
<td>Operands equal so far; R1 and R2 point to next bytes to process</td>
</tr>
</tbody>
</table>

- 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
Translate Extended Instruction

- TRE is similar to TR, but more flexible:
  1. Translated string address is the R₁ operand, translate table address is the R₂ operand
     - The string length is in (odd register) R₁+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:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All bytes translated; R₁ incremented by length, R₁+1 set to 0</td>
</tr>
<tr>
<td>1</td>
<td>R₁ points to the byte matching the stop character; R₁+1 decremented by the number of bytes processed before the match</td>
</tr>
<tr>
<td>3</td>
<td>R₁ incremented and R₁+1 decremented by the number of bytes processed</td>
</tr>
</tbody>
</table>

Notes

Compare Until Substring Equal Instruction (*)

- CUSE searches for common substrings of a specified length
- The matching substrings must be at the same offset in both strings
  `‘ABCDEF’` 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 R₁ and R₂; Operand lengths are in corresponding odd-numbered registers R₁+1 and R₂+1
  2. The rightmost bytes of GR0 and GR1 contain the desired substring length and the padding byte, respectively

- Condition Code settings:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Equal substrings found; R₁, R₂, and lengths updated; or, the substring length is 0, and R₁, R₂ are unchanged</td>
</tr>
<tr>
<td>1</td>
<td>Ended at longer operand, last bytes were equal (allows continuing search for further matches if required)</td>
</tr>
<tr>
<td>2</td>
<td>Ended at longer operand, last bytes were unequal; or, both operand lengths = 0 and the substring length is &gt; 0</td>
</tr>
<tr>
<td>3</td>
<td>Search incomplete, last compared bytes unequal; R₁, R₂, lengths are updated</td>
</tr>
</tbody>
</table>

Notes
Summary

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Length control</th>
<th>End-char control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>MVCL</td>
<td>MVST</td>
</tr>
<tr>
<td></td>
<td>MVCLE</td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>CLCL</td>
<td>CLST</td>
</tr>
<tr>
<td></td>
<td>CLCLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CUSE</td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td>SRST</td>
</tr>
<tr>
<td>Translate</td>
<td></td>
<td>TRE</td>
</tr>
</tbody>
</table>

- 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

Notes

Other Types of Character Data

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

Notes
**Character Representations**

- 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

**EBCDIC Representations and Code Pages**

- 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 alphabets, and
  - + - * % & ’ ( ) _ . / : ; ?
  - It does not include # @ $ (allowed in Assembler Language symbols)
- All modern EBCDIC code pages support the “euro” character €
**ASCII**

- 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 EBCDIC Data**

- 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
  ```
  │ sb │ sb │ SO │ db │ db │ db │ db │ db │ db │ SI │ sb │ sb │ SO │ db │ db │ db │ db │ SI │ sb │ sb │
  └───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
  3 DB characters   2 DB chars
  ```
- 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+00FF 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’

Unicode Instructions

- We will discuss the instructions in three groups:
  1. String search, compare, and move instructions
  2. Translation instructions
  3. Format conversion instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRSTU</td>
<td>Search String Unicode</td>
<td>CLCLU</td>
<td>Compare Logical Long Unicode</td>
</tr>
<tr>
<td>MVCLU</td>
<td>Move Long Unicode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TROO</td>
<td>Translate One to One</td>
<td>TROT</td>
<td>Translate One to Two</td>
</tr>
<tr>
<td>TRTO</td>
<td>Translate Two to One</td>
<td>TRTT</td>
<td>Translate Two to Two</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU12, CUFFU</td>
<td>Convert UTF-8 to UTF-16</td>
<td>CU14</td>
<td>Convert UTF-8 to UTF-32</td>
</tr>
<tr>
<td>CU21, CUUUF</td>
<td>Convert UTF-16 to UTF-8</td>
<td>CU24</td>
<td>Convert UTF-16 to UTF-32</td>
</tr>
<tr>
<td>CU41</td>
<td>Convert UTF-32 to UTF-8</td>
<td>CU42</td>
<td>Convert UTF-32 to UTF-16</td>
</tr>
</tbody>
</table>

Notes
Unicode Search, Move, and Compare

- 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 R₂ for a byte pair matching the rightmost 2 bytes of GR₀; R₁ has the address of the first byte after the string
  - R₂ incremented by 2 for each comparison
- **MVCLU**: moves pairs of bytes from area addressed by R₃ to area addressed by R₁; lengths in R₁₊₁, R₃₊₁; “padding pair” is low-order 16 bits of D₂(B₂) 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

Optional Operands

- 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

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>//</td>
<td>//</td>
<td>R₁</td>
<td>R₂</td>
</tr>
<tr>
<td>RRE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>M₃</td>
<td>//</td>
<td>R₁</td>
<td>R₂</td>
<td></td>
</tr>
<tr>
<td>RRF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assembler instruction format:
  mnemonic R₁,R₂[,M₃] [ ] indicates optional operand
### Unicode Translation

- 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 double-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!

### Conversion Among Transformation Formats

- 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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU12, CUTFU</td>
<td>Convert UTF-8 to UTF-16</td>
<td>CU14</td>
<td>Convert UTF-8 to UTF-32</td>
</tr>
<tr>
<td>CU21, CUUTF</td>
<td>Convert UTF-16 to UTF-8</td>
<td>CU24</td>
<td>Convert UTF-16 to UTF-32</td>
</tr>
<tr>
<td>CU41</td>
<td>Convert UTF-32 to UTF-8</td>
<td>CU42</td>
<td>Convert UTF-32 to UTF-16</td>
</tr>
</tbody>
</table>

- Operand formats:
  - $CUxx \ R_1, R_2, [M_3]$ for CU12, CU14, CU21, CU24
  - $CUxx \ R_1, R_2$ for CU41, CU42

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

- Each instruction uses an optional operand to create six "additional" instructions with a single opcode

Operand format:

Mnemonic \( R_{1}, R_{2}, [M_{3}] \) \( M_{3} \) bits are \( 8'\text{AFLO}' \)

- \( M_{3} \) mask bits:
  - \( A \): Argument characters are 1 byte
    - 0: Argument characters are 1 byte
    - 1: Argument characters are 2 bytes
  - \( F \): Function codes are 1 byte
    - 0: Function codes are 1 byte
    - 1: Function codes are 2 bytes
  - \( L \): Full range of argument and function codes allowed
    - 0: Full range of argument and function codes allowed
    - 1: Argument > 255 means function code assumed to be zero

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

Notes

---

Byte Reversal and Workstation Data

- Suppose a 32-bit integer \( X'12345678' \) starts at address \( X'2400' \).
  
<table>
<thead>
<tr>
<th>12</th>
<th>34</th>
<th>56</th>
<th>78</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td>2403</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IBM System z ("Big-Endian")

- On some processors (e.g. Intel) it's stored like this:
  
<table>
<thead>
<tr>
<th>78</th>
<th>56</th>
<th>34</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td>2403</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some workstations ("Little-Endian")

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRV</td>
<td>Load Reversed (32)</td>
<td>LRVG</td>
<td>Load Reversed (64)</td>
</tr>
<tr>
<td>LRVH</td>
<td>Load Halfword Reversed (16)</td>
<td>LRVGR</td>
<td>Load Register Reversed (64)</td>
</tr>
<tr>
<td>STRV</td>
<td>Store Reversed (32)</td>
<td>STRVH</td>
<td>Store Halfword Reversed (16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STRVG</td>
<td>Store Reversed (64)</td>
</tr>
</tbody>
</table>

Notes
This chapter explores 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.

### Notes

- 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.

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>Move Numerics</td>
<td>MVZ</td>
<td>Move Zones</td>
</tr>
<tr>
<td>PACK</td>
<td>Pack</td>
<td>UNPK</td>
<td>Unpack</td>
</tr>
<tr>
<td>PKA</td>
<td>Pack ASCII</td>
<td>UNPKA</td>
<td>Unpack ASCII</td>
</tr>
<tr>
<td>PKU</td>
<td>Pack Unicode</td>
<td>UNPKU</td>
<td>Unpack Unicode</td>
</tr>
</tbody>
</table>

- All these instructions have SS-1 or SS-2 format.
Zoned Decimal Representation

- 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)

```
 ┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐  Z  n  Z  n  Z  n  Z  n  Z  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  d  Z  d  Z  d  Z  d  S  d
 └───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘  Z=zone digit, d=decimal digit, S=sign code
```

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

Zoned Decimal Constants

- 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 ZLS'29'       Generates X'F0F0F0F2C9'
```

- Only a length modifier is valid (no integer, scale, exponent)
- Decimal points in nominal values are ignored
- The Assembler assigns Integer and Scale attributes
  - But the meaning of a decimal point is up to you!

Notes
Packed Decimal Representation

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)

```
  ┌───┬───┬───┬───┬───┬───┐
  │ d d │ d d │ d d │ d d │ d S │
  └───┴───┴───┴───┴───┴───┘
```

- 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)

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+12345</td>
<td>X'12345C'</td>
</tr>
<tr>
<td>-0012345</td>
<td>X'0012345D'</td>
</tr>
<tr>
<td>+3</td>
<td>X'3C'</td>
</tr>
<tr>
<td>-09990</td>
<td>X'09990D'</td>
</tr>
<tr>
<td>3</td>
<td>X'039C'</td>
</tr>
</tbody>
</table>

Packed Decimal Constants

- Defined with constant type P

<table>
<thead>
<tr>
<th>Symbol</th>
<th>DC</th>
<th>Value</th>
<th>Generated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon1</td>
<td>DC</td>
<td>P'12345'</td>
<td>X'12345C'</td>
</tr>
<tr>
<td>PCon2</td>
<td>DC</td>
<td>P'-27,+62'</td>
<td>X'047D062C'</td>
</tr>
<tr>
<td>PCon3</td>
<td>DC</td>
<td>PL'999'</td>
<td>X'0000999C' (Padded on left)</td>
</tr>
<tr>
<td>PCon4</td>
<td>DC</td>
<td>PL'12345'</td>
<td>X'345C' (Truncated on left)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>DC</th>
<th>Value</th>
<th>Generated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon5</td>
<td>DC</td>
<td>P'1234.5'</td>
<td>X'12345C'</td>
</tr>
<tr>
<td>PCon6</td>
<td>DC</td>
<td>P'1.2345'</td>
<td>X'12345C'</td>
</tr>
</tbody>
</table>

- HLASM assigns Integer and Scale attributes:
  - PCon5: Integer attribute = 4, Scale attribute = 1
  - PCon6: Integer attribute = 1, Scale attribute = 4

Notes

---

Packed Decimal Constants

- Defined with constant type P

<table>
<thead>
<tr>
<th>Symbol</th>
<th>DC</th>
<th>Value</th>
<th>Generated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon1</td>
<td>DC</td>
<td>P'12345'</td>
<td>X'12345C'</td>
</tr>
<tr>
<td>PCon2</td>
<td>DC</td>
<td>P'-27,+62'</td>
<td>X'047D062C'</td>
</tr>
<tr>
<td>PCon3</td>
<td>DC</td>
<td>PL'999'</td>
<td>X'0000999C' (Padded on left)</td>
</tr>
<tr>
<td>PCon4</td>
<td>DC</td>
<td>PL'12345'</td>
<td>X'345C' (Truncated on left)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>DC</th>
<th>Value</th>
<th>Generated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon5</td>
<td>DC</td>
<td>P'1234.5'</td>
<td>X'12345C'</td>
</tr>
<tr>
<td>PCon6</td>
<td>DC</td>
<td>P'1.2345'</td>
<td>X'12345C'</td>
</tr>
</tbody>
</table>

- HLASM assigns Integer and Scale attributes:
  - PCon5: Integer attribute = 4, Scale attribute = 1
  - PCon6: Integer attribute = 1, Scale attribute = 4

Notes

---
Converting Between Packed and Zoned

- Use the PACK and UNPK instructions
- Both have Assembler Language syntax
  \[ \text{mnemonic} \quad D_1(N_1,B_1),D_2(N_2,B_2) \]
- Machine instruction format:
  \[
  \begin{array}{c|c|c|c|c|c}
    \text{opcode} & L_1 & B_1 & D_1 & B_2 & D_2 \\
  \end{array}
  \]
- Encoded Lengths L are one less than Program Lengths N
- Each operand can take one of four forms:

<table>
<thead>
<tr>
<th></th>
<th>Implied Length</th>
<th>Explicit Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied Address</td>
<td>( S_1 )</td>
<td>( S_1(N) )</td>
</tr>
<tr>
<td>Explicit Address</td>
<td>( D_1(B_1) )</td>
<td>( D_1(N_1,B_1) )</td>
</tr>
</tbody>
</table>

Notes

The PACK Instruction

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

\[
\text{PACK} \quad \text{Target,Source}
\]

- 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!

Notes
The UNPK Instruction

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

\[
\text{UNPK Target,Source}
\]

\[
\begin{array}{cccccc}
\text{d d} & \text{d d} & \text{d S} & \text{Packed source (second) operand} \\
\hline
\text{Z d} & \text{Z d} & \text{Z d} & \text{Z d} & \text{S d} & \text{Zoned target (first) operand}
\end{array}
\]

- 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!

Notes
### Printing Hexadecimal Values

- 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:
     ```assembly
     MVC WorkArea+4(4),SourceData
     WorkArea DS CL8,X         ← The extra byte is important!
     ```
  2. Unpack one extra byte (at the right end):
     ```assembly
     UNPK WorkArea(9),WorkArea+4(5) Extra byte is swapped
     ```
  3. Translate the “spread hex” to EBCDIC characters
     ```assembly
     TR WorkArea,=C'0123456789ABCDEF'−C'0'
     ```
  4. 8 bytes at WorkArea are ready for display or print

---

### Section 28: Packed Decimal Arithmetic Overview

- Usually gives expected results
  - Operand size limitations for some operations
- Notation corresponds to internal representations
  ```text
  Packed: 12 34 56 7D written 1234567–
  Zoned: F1 F2 F3 C4 written 1234+
  ```
- Zoned values must be converted to packed for arithmetic

---

Notes

---

Notes
Packed Decimal Arithmetic: General Rules

- 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

Decimal Addition and Subtraction

- 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:

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Result is zero</td>
</tr>
<tr>
<td>1</td>
<td>Result is less than 0</td>
</tr>
<tr>
<td>2</td>
<td>Result is greater than 0</td>
</tr>
<tr>
<td>3</td>
<td>Decimal overflow</td>
</tr>
</tbody>
</table>

- Advice: avoid operand overlap

Notes
Decimal Comparison

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

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand 1 = Operand 2</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 &lt; Operand 2</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 &gt; Operand 2</td>
</tr>
</tbody>
</table>

Notes

---

Decimal Multiplication

- 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

Notes

---

124
Decimal Division

Before:  

\[ \text{dividend} \]  

After:  

\[ \text{quotient} \text{ remainder} \]  

\[ \text{divisor} \]  

\[ \text{divisor} \]  

- The remainder has the same byte 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) \( \leq \) 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

Section 29: Packed Decimal Instructions

- The packed decimal instructions are

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Add Decimal</td>
<td>SP</td>
<td>Subtract Decimal</td>
</tr>
<tr>
<td>MP</td>
<td>Multiply Decimal</td>
<td>DP</td>
<td>Divide Decimal</td>
</tr>
<tr>
<td>CP</td>
<td>Compare Decimal</td>
<td>ZAP</td>
<td>Zero and Add Decimal</td>
</tr>
<tr>
<td>SRP</td>
<td>Shift and Round Decimal</td>
<td>MVO</td>
<td>Move with Offset</td>
</tr>
<tr>
<td>TP</td>
<td>Test Decimal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{opcode} \quad L1 \quad L2 \quad B1 \quad D1 \quad B2 \quad D2
\]

- All instructions process operands from right to left

Notes
Test Decimal (TP) Instruction

- TP tests the validity of its operand. Assembler Language syntax:
  \[ \text{TP } D_i(N, B_j) \]
- Its machine instruction format differs from the other instructions:

\[
\begin{array}{cccccccc}
\text{opcode} & L & /// & B1 & D1 & ///// & \text{opcode} \\
\end{array}
\]
- Valid Assembler Language instruction operand formats:

<table>
<thead>
<tr>
<th>Explicit Length</th>
<th>Implied Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Address</td>
<td>( D_i(N, B_j) )</td>
</tr>
<tr>
<td>Implied Address</td>
<td>( S(N) )</td>
</tr>
</tbody>
</table>

- Condition Code settings:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All digit codes and the sign code are valid.</td>
</tr>
<tr>
<td>1</td>
<td>The sign code is invalid.</td>
</tr>
<tr>
<td>2</td>
<td>At least one digit is invalid.</td>
</tr>
<tr>
<td>3</td>
<td>The sign code and at least one digit are invalid.</td>
</tr>
</tbody>
</table>

Zero and Add (ZAP) Instruction

- 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
  \[ \text{ZAP } D_i(N, B_j), D_j(N, B_i) \text{ or } \text{Target}(N_i), \text{Source}(N_j) \]
- Condition Code settings

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Result is zero.</td>
</tr>
<tr>
<td>1</td>
<td>Result is less than zero.</td>
</tr>
<tr>
<td>2</td>
<td>Result is greater than zero.</td>
</tr>
<tr>
<td>3</td>
<td>Decimal overflow.</td>
</tr>
</tbody>
</table>
Add Decimal (AP) and Subtract Decimal (SP) Instructions

- The result replaces the first operand
- Assembler Language syntax:
  \[
  \text{AP } D_1(N_1, B_1), D_2(N_2, B_2) \hspace{1cm} \text{(Same for SP)}
  \]
- Condition Code settings:

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Result is zero.</td>
</tr>
<tr>
<td>1</td>
<td>Result is less than zero.</td>
</tr>
<tr>
<td>2</td>
<td>Result is greater than zero.</td>
</tr>
<tr>
<td>3</td>
<td>Decimal overflow.</td>
</tr>
</tbody>
</table>

- Overflow is possible if the first operand is too short

<table>
<thead>
<tr>
<th>AP</th>
<th>P123, P9</th>
<th>Result at P123 = 132+. CC=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>P9,P234</td>
<td>Result at P9 = 3+, CC=1 (overflow)</td>
</tr>
<tr>
<td>P123</td>
<td>DC P′+123′</td>
<td></td>
</tr>
<tr>
<td>P234</td>
<td>DC P′+234′</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>DC P′+9′</td>
<td></td>
</tr>
</tbody>
</table>

Notes

Compare Decimal (CP) Instruction

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

<table>
<thead>
<tr>
<th>CC</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operands are equal.</td>
</tr>
<tr>
<td>1</td>
<td>First operand is low.</td>
</tr>
<tr>
<td>2</td>
<td>First operand is high.</td>
</tr>
</tbody>
</table>

- Examples:
  \[
  \text{CP } =\text{P′+5′}, =\text{P′+3′} \hspace{1cm} \text{CC}=2
  \]
  \[
  \text{CP } =\text{P′+3′}, =\text{P′+5′} \hspace{1cm} \text{CC}=1
  \]
  \[
  \text{CP } =\text{P′+0′}, =\text{P′+0′} \hspace{1cm} \text{CC}=0
  \]

Notes
Multiply Decimal (MP) Instruction

- Assembler Language syntax:
  \[\text{MP} \ D_1(N_1,B_1),D_2(N_2,B_2)\]

- Operand length restrictions:
  \[2 \leq N_1 \leq 16, \quad 1 \leq N_2 \leq 8, \quad N_1 > N_2\]
  \[1 \leq L_1 \leq 15, \quad 0 \leq L_2 \leq 7, \quad L_1 > L_2\]

- 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

  ┌────────────────────────┬──────────────┐
  │000 ──────────────── 000 aaaaaaaaaaaaas│ First operand (multiplicand)
  ├────────────────────────┼──────────────┘
  │bbbbbbbbbbbbbbbbbbbbbbbs│ Second operand (multiplier)
  └────────────────────────┘

Notes

Divide Decimal (DP) Instruction

- Assembler Language syntax:
  \[\text{DP} \ D_1(N_1,B_1),D_2(N_2,B_2)\]

- Like binary division, the quotient and remainder replace the dividend (but in the opposite order)

  ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
  │ dividend │ quotient │ divisor │
  └───────────┘ └───────────┘ └───────────┘

  - 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:
  \[2 \leq N_1 \leq 16, \quad 1 \leq N_2 \leq 8, \quad N_1 > N_2\]
  \[1 \leq L_1 \leq 15, \quad 0 \leq L_2 \leq 7, \quad L_1 > L_2\]

- Example:
  \[\text{ZAP} \ Dvnd,=P'162843'\]  \[\text{Initialize dividend}\]
  \[\text{DP} \ Dvnd,=P'762'\]  \[\text{Divide by 762}\]
  \[Dvnd \ DS \ PL4\]  \[\text{Result: } X'213C537C'\]

Notes
SRP Instruction

• SRP multiplies and divides by a power of 10, with optional quotient rounding.
• Assembler Language syntax:
  \[ \text{SRP } D_1(N_1,B_1),D_2(B_2),I_3 \]
• Machine instruction format:

\[
\begin{array}{cccccc}
F_0 & L_1 & I_3 & B_1 & D_1 & B_2 & D_2 \\
\end{array}
\]

• Shift amount and direction determined by low-order 6 bits of second-operand Effective Address:
  \[ B'100000' = -32 \leq \text{shift count} \leq +31 = B'011111' \]
• Examples:
  \[
  \begin{align*}
  \text{SRP } X,3,0 & \quad \text{Multiply operand at } X \text{ by 1000} \\
  \text{SRP } X,64-3,5 & \quad \text{Divide operand at } X \text{ by 1000, round last digit}
  \end{align*}
  \]
• Possible overflow on left shifts
• Rounded results are slightly biased

Move With Offset (MVO) Instruction

• MVO moves the second operand to the first, but offset to the left by 4 bits.
• Assembler Language syntax:
  \[ \text{MVO } D_1(N_1,B_1),D_2(N_2,\&B_2) \]
• Example of two 4-byte operands with signs s1, s2:

\[
\begin{array}{cccccccc}
\text{x x x x} & \text{x x x} & \text{x} & \text{s}1 & \text{Operand 1 before} \\
\text{b c d e f g h} & \text{s}2 & \text{Operand 2 offset left 4 bits} \\
\text{c d e f g h s}2 & \text{s}1 & \text{Operand 1 after}
\end{array}
\]
• If \( N_2 \geq N_1 \), high-order digits are lost
• If \( N_2 < N_1 \), high-order digits positions are filled with zeros
Decimal Shifting Using MVO (*)

- Before SRP was available, MVO was used for packed 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

Notes

Scaled Packed Decimal Computations: General Rules

- 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 = I+F, and Value = I.F
- Given operands 1 and 2 (I,F₁ and I₂,F₂), then:
  - Sum or Difference I,F: I=Max(I₁,I₂), F=Max(F₁,F₂)
  - Product I,F: I=I₁+I₂, F=F₁+F₂
  - Quotient I,F: I=I₁+I₂; for an N-digit result, F=N−I
- You must keep these in mind doing scaled packed decimal arithmetic

Notes
Section 30: Converting and Formatting Packed Decimal Data

- 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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVB</td>
<td>Convert to Binary (32)</td>
<td>CVD</td>
<td>Convert to Decimal (32)</td>
</tr>
<tr>
<td>CVBY</td>
<td>Convert to Binary (32)</td>
<td>CVDY</td>
<td>Convert to Decimal (32)</td>
</tr>
<tr>
<td>CVBG</td>
<td>Convert to Binary (64)</td>
<td>CVDG</td>
<td>Convert to Decimal (64)</td>
</tr>
<tr>
<td>ED</td>
<td>Edit</td>
<td>EDMK</td>
<td>Edit and Mark</td>
</tr>
</tbody>
</table>

- 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

Notes

CVD, CVDY, and CVDG Instructions

- 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
    
    \[
    \begin{align*}
    \text{CVD} 7, \text{WorkArea} & \quad \text{Convert to packed decimal at WorkArea} \\
    \text{CVDY} 7, \text{WorkArea} & \quad \text{Convert to packed decimal at WorkArea} \\
    \text{WorkArea DS D} & \quad \text{Result} = X'00000000 0000135' \\
    \end{align*}
    \]

- Example: suppose c(GG8) = X’4000000000000000’ = 2^{62}
  - CVDG converts 64-bit integers to 16 bytes of 31 packed decimal digits.
    
    \[
    \begin{align*}
    \text{CVDG} 8, \text{WrkArea2} & \quad \text{Convert to 16-byte packed decimal} \\
    \text{WrkArea2 DS 2D} & \quad \text{Result} = X'00000000 0000065 68601842 7387904' \\
    \end{align*}
    \]

- The operands need not be aligned on any specific boundary
  - But doubleword alignment can improve performance

Notes
CVB, CVBY, and CVBG Instructions

- Convert 8- or 16-byte packed decimal data to 2’s complement binary in a register
  
  ```
  CVB 0,PACKNUM RESULT IN G0 = X’FFFFFFFF’ = -135
  CVB 1,PACKNUM RESULT IN G0 = X’FFFFFF87F22087’
  ```
  
  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

Notes

Editing Overview

- Assembler Language syntax for ED and EDMK:
  ```
  mnem D1(N,B1),D2(B2) or Pattern(N),PackData
  ```

- Machine instruction format:

<table>
<thead>
<tr>
<th>opcode</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
</table>

- The basic operation of the instructions:

  ```
  dddd.ddddds → edit → C C C C C C C C
  ```

  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 Overview (continued)

- 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 \texttt{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 \texttt{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 \texttt{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’•\texttt{ddd,dsd}’

Notes

---

Editing Overview (continued)

- 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:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All source digits 0, or no digit selectors in pattern</td>
</tr>
<tr>
<td>1</td>
<td>Nonzero source digits, and SI is ON (result &lt; 0)</td>
</tr>
<tr>
<td>2</td>
<td>Nonzero source digits, and SI is OFF (result &gt; 0)</td>
</tr>
</tbody>
</table>

Notes
Simple Examples of Editing

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'+++++++d'
   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

Notes

Single-Field Editing

• 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',.X'20206820',X'2120' Pattern = C'+++++++d'
   LineX DS CL(LineX) Edited result here, C'+1234567890'

• 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 F'0012345' Credit balance of $123.45
   Pat2 DC C',.X'202068202020214B2020'.C CREDIT Pattern = C'+++++++d'
   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 CREDIT

Notes
The EDMK Instruction

- 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′202068202021482020′.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

Notes

Editing Multiple Fields (*)

- 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′402021204022202120′.C+dsf+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

Notes
These are complex but powerful instructions

<table>
<thead>
<tr>
<th>Pattern Character (PC)</th>
<th>Get Source Digit?</th>
<th>SI</th>
<th>Source Digit</th>
<th>Result</th>
<th>Set SI</th>
<th>Sign code in right digit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'20' (DS)</td>
<td>Yes</td>
<td>1</td>
<td>Any</td>
<td>ZD</td>
<td>1</td>
<td>+, set SI OFF; otherwise, leave it unchanged</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Nonzero</td>
<td>ZD</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Zero</td>
<td>Fill Char</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>X'21' (SS)</td>
<td>Yes</td>
<td>1</td>
<td>Any</td>
<td>ZD</td>
<td>1</td>
<td>+, set SI OFF; otherwise, leave it unchanged</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Nonzero</td>
<td>ZD</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Zero</td>
<td>Fill Char</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>X'22' (FS)</td>
<td>No</td>
<td>—</td>
<td>—</td>
<td>Fill Char</td>
<td>0</td>
<td>No source byte is examined</td>
</tr>
<tr>
<td>Other (MC)</td>
<td>No</td>
<td>0</td>
<td>—</td>
<td>Fill Char</td>
<td>0</td>
<td>No source byte is examined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>—</td>
<td>MC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Fill Character (FC)</td>
<td>No</td>
<td>N/A</td>
<td>—</td>
<td>Fill Char</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Notes
Simple I/O Macros

- 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

Notes

Notation and Terminology Conventions

- 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

Notes
The CONVERTI Macro Instruction

- **CONVERTI** is written
  \[
  \text{CONVERTI } \langle\text{number}\rangle,\langle\text{address}\rangle[,\text{ERR}=<\text{address}>][,\text{STOP}=<\text{address}>]
  \]
  - The digits starting at the second operand \(\langle\text{address}\rangle\) are converted to binary in the general register designated by \(\langle\text{number}\rangle\)
    - 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 \(\langle\text{number}\rangle\) 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:**
  \[
  \text{CONVERTI } 3,\text{Data } c(\text{GR}3) = \text{X}'00000013' \quad \text{c}(\text{GR}1) = \text{A(\text{Data}+3)} \quad ('?' \quad \text{Data}\quad \text{DC}\quad C'\text{+019?}')
  \]

### Notes

---

The CONVERTO Macro Instruction

- **CONVERTO** is written
  \[
  \text{CONVERTO } \langle\text{number}\rangle,\langle\text{address}\rangle
  \]
  - The contents of the register specified by \(\langle\text{number}\rangle\) (not the \(\langle\text{number}\rangle\) itself!) are converted to \(N\) characters in memory starting at \(\langle\text{address}\rangle\)
    - \(0 \leq \langle\text{number}\rangle \leq 15: N=12\)
    - \(16 \leq \langle\text{number}\rangle \leq 31: N=21\)
    - \(32 \leq \langle\text{number}\rangle \leq 47: N=20\)
  - The first character of the result is always a blank
  - If the value of \(\langle\text{number}\rangle\) 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):
  \[
  \begin{align*}
  \text{CONVERTO } 4,\text{GR4Val} & \quad \text{c(\text{GR4Val})} = \text{X}'-2147483648' \\
  \text{CONVERTO } 22,\text{GG6Val} & \quad \text{c(\text{GG6Val})} = \text{X}'-9223372036854775808' \\
  \text{CONVERTO } 34,\text{FR2Val} & \quad \text{c(\text{FR2Val})} = \text{X}'\text{FEDCBA9876543210}'
  \end{align*}
  \]

### Notes

---
The DUMPOUT Macro Instruction

- **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 0001A20 4 F001A002 00000006 *....0..0........S.0.......*
01A020 98EFE000 070090EF F03058F0 F03805EF 00F12802 0001A000 0 8001A22C 0001A026 *Q.......0..0........S.....*
```

Notes

---

The PRINTLIN Macro Instruction

- **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'"Title for Top Line of a Page'"
  ```

Notes
The PRINTOUT Macro Instruction

- PRINTOUT supports 3 types of operand: <name>, <number>, and *
  
  - <nfs> PRINTOUT [<name>...][<number>...]  
  - 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:
  
  ```plaintext
  PRINTOUT 1,19,32,*  Print GR1, GG3, FPR0, terminate
  ```

- Produces output like this:
  
  ```plaintext
  *** PRINTOUT requested at Address 01A132, Statement 808, CC=0  
  GPR 1 = X'0001A197' = 106903  
  GGR 3 = X'FFFFFFFFFFFFFFF' = -1  
  FPR 0 = X'0000000000000000'  
  *** Execution terminated by PRINTOUT * at Address 01A132
  ```

Notes

The READCARD Macro Instruction

- READCARD reads 80-byte records into your program
  
  ```plaintext
  <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

  ```plaintext
  *** Execution terminated by Reader EOF
  ```

- Example:
  
  ```plaintext
  GetARec READCARD MyRecord,EndFile  
  EndFile -- Do something about no more records
  ```

Notes
• 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,*
Sample Program

- 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
LA 1,(0,1) INCREMENT CARD COUNTER
PRINTOUT 1 PRINT THE COUNTER REGISTER
PRINTLIN OUT, LINELEN PRINT A LINE
CONVERGI 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
ADD 3,3 SET GG3 TO 0
BCTR 3,0 NON SET GG3 TO -1
PRINTOUT 1, 19, 32,* PRINT GR1, GG3, FPDR, TERMINATE
OUT DC C\'INPUT RECORD \'' CARD IMAGE HERE
LNLINER EQU \'' OUT \'' define line length
OUTDATA DS CL12 CONVERTED CHARACTERS
LAST EQU * LAST BYTE OF PROGRAM
END
```

Notes