Assembler Language Programming

for

IBM System z™ Servers

Lecture Slides

Version 2.00, Chapters I to VIII

John R. Ehrman

IBM Silicon Valley Lab
ehrman@us.ibm.com

---

Note:

Slides are keyed in their bottom left corner to the text, referring to the related Chapter and Section.
Second Edition (March 2016)

IBM welcomes your comments. Please address them to

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

After June 1, 2016, please address them to

john.ehrman@comcast.net

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

- The major divisions of the text are:
  
  **Chapter I:** Number representations, arithmetic, and base conversions  
  **Chapter II:** CPU, registers, PSW, and instructions  
  **Chapter III:** Fundamentals of Assembler Language  
  **Chapter IV:** Defining data and work areas  
  **Chapter V:** Basic instructions using General Registers; branches  
  **Chapter VI:** Addressing, loop instructions, immediate operands  
  **Chapter VII:** Bit and character data, instructions, and representations  
  **Chapter VIII:** Packed decimal data and instructions  
  **Chapter IX:** Floating-point data and instructions  
  **Chapter X:** Large programs, modularization, and subroutines  
  **Chapter XI:** Dummy control sections and enhanced USING statements  
  **Chapter XII:** System services, exception handling, reenterability, recursion  
  **Appendix A:** Conversion and reference tables  
  **Appendix B:** Useful macros for conversion, reading, display and printing
Programming Environments

- Programming Environments
  - We assume your programs will execute on one of IBM’s z/OS (MVS), z/VM (CMS), or z/VSE operating systems
    - The simple macros in Appendix B have worked on all three

- Some section headings end with “(*)”
  - These may be more detailed or difficult topics

- The Exercises and Programming Problems are strongly recommended
  - Their numbers are followed by a parenthesized digit with an estimated difficulty from (1) = easy, to (5) = difficult
  - Recommended exercises and problems are tagged with “+”
Introduction

- A computer can be viewed at many levels, such as
  - A collection of logical circuits
  - Techniques used to make circuits perform operations like addition, division
  - Instructions to perform an operation like addition, data movement
  - A processor to evaluate mathematical expressions, maintain data
  - A simulator of physical processes: traffic flow, weather prediction

- Our concern is mainly the middle level
  - With occasional excursions into neighboring levels

- The assembler described is IBM’s “IBM High Level Assembler for z/OS & z/VM & z/VSE”, known as “HLASM”
Why Program in Assembler Language (and Why Not)?

Why?

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

Why not?

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

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

- It’s dead!
  Many organizations have major investments in Assembler Language applications that provide functionality and speed; frequent updates are a business necessity.

- It’s hard to learn!
  The language is very simple; difficulties can be due to programming style or instruction unfamiliarity.

- It’s hard to maintain!
  Research has shown there’s little difference among languages in maintenance costs; “clean” (and messy) programs can be written in any language.

- Converting to another language is easy!
  This is very rarely true, and attempts can be very expensive.

- You can’t do structured programming!
  The HLASM “Toolkit Feature” has a powerful set of Structured Programming Macros.
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
Notation and Terminology

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

```
   4  4 ─── Field widths
  ┌───┬───┐
  │ 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
What’s an “Operand”?  

- 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 \( c(\text{SaveArea}) \) are all operands that are subjected to an operation.

- The intended meaning is usually clear from context
Section 2 describes fundamentals of number representations:

- Binary and hexadecimal numbers, and positional notation
- Conversion among different representations
- Logical (unsigned) and arithmetic (signed) representations
- Two’s complement (signed) representation
- Binary addition and subtraction; overflow; signed vs. unsigned results
- Alternative representations of signed binary values
We’ll start with integer values (no fractional parts):

- Decimal integer values like 1705 mean \(1000 + 700 + 00 + 5\), or \(1 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0\) (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′nnn′
  - 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>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

- Hexadecimal numbers are written X’n.nn’n’
  - 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
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
Two’s Complement

- Binary addition is very simple:

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+0 & +1 & +0 & +1 \\
0 & 1 & 1 & 10 \text{ (carry)}
\end{array}
\]

- 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
      \(+1\)
      \(01111111\) (signed +127) \(11111111\) (signed -1)
    - \(11111101\) (signed -3) \(00011111\) (signed +31)
      \(+1\)
      \(00000010\) ones’ complement \(11100000\) ones’ complement
      \(+1\)
      \(00000011\) (signed +3) \(11100001\) (signed -31)

  - Carries out of the leftmost bit: ignored for unsigned, important for signed
  - But most arithmetic instructions take note of carries
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 \(-8 \leq \text{value} \leq +7\)):
    
    \[
    \begin{array}{c|c|c|c}
    1111 & 0010 & 0100 \\
    \hline
    (-1) & (+2) & (+4) \\
    \hline
    +0001 & +0010 & +0100 \\
    (+1) & (+2) & (+4) \\
    \hline
    0000 & 0100 & 1000 \\
    (+0) & (+4) & (-8, overflow) \\
    \end{array}
    \]
  - 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 \leq \text{value} \leq 15):
    
    \[
    \begin{array}{c|c|c|c}
    1111 & 0010 & 1100 \\
    \hline
    (15) & (2) & (12) \\
    \hline
    +0001 & +0010 & +1001 \\
    (1) & (2) & (9) \\
    \hline
    0000 & 0100 & 0101 \\
    (0, carry) & (4, no carry) & (5, carry) \\
    \end{array}
    \]

- Conditional branch instructions (described in Section 15) can test for overflow (arithmetic addition or subtraction) and carries (logical addition or subtraction)
• Subtraction is slightly more complicated than addition...
  1. Form the ones’ complement of the second (subtrahend) operand
  2. Add the first (minuend) and complemented second operands \textit{and} a low-order 1 bit (but in a \textit{single} operation!)

  Examples, using signed 4-bit values:
  \begin{align*}
  -1 - (+1) & \quad +2 - (+2) & \quad 3 - 5 \\
  1111 (-1) & \quad 0010 (+2) & \quad 0011 (+3) \\
  1110 (+1,\text{comp}) & \quad 1101 (-2,\text{comp}) & \quad 1010 (+5,\text{comp}) \\
  +1 & \quad +1 & \quad +1 \\
  1110 (-2, \text{carry}) & \quad 0000 (+0, \text{carry}) & \quad 1110 (-2, \text{no carry})
  \end{align*}

  - Arithmetic subtraction: overflows possible; logical subtraction: carries are noted

• Adding the first operand directly to the two’s complement of the second operand works \textit{almost}, but \textit{not} all the time!
• Why must we add all three items at once?

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

\[
\begin{array}{c c}
+1-(-8) \text{ right way} & +1-(-8) \text{ wrong way} \\
0001 (+1) & 0001 (+1) \\
0111 (-8, 1s \text{ comp}) & +1000 (-8, 2’s \text{ comp}) \\
+ & + \\
1001 (-7, \text{ overflow}) & 1001 (-7, \text{ no overflow})
\end{array}
\]

- Overflow occurred in forming the two’s complement of $-8$ before adding

• Adding all three items at once guarantees correct overflow detection
A Circular Representation of 4-bit Signed Integers

- A circular representation of 4-bit signed integers:

  o 0100
  o 0101 o 0011
  o 0010

  x overflow point

  0001 o 0010
  0000

  carry point x

  1011 o 1111
  1100 o 1110
  1101

o = positive number,
• = negative number.

Addition: move counter-clockwise

Subtraction: move clockwise

Overflow: move past the overflow point

Carry from high-order bit: move past the carry point
Logical vs. Arithmetic Results, Other Representations

- 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>
Exercise Answers

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

• Slide 6:
  - X’91’, X’1F4’
  - 1891, 247
  - B’11101100011’, B’11110111’

• Slide 7:
  - X’929’, 65607, 10B513.

• Slide 8:
  - -86, 170
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>8DF</th>
<th>8E0</th>
<th>8E1</th>
<th>8E2</th>
<th>8E3</th>
<th>8E4</th>
<th>8E5</th>
<th>8E6</th>
<th>8E7</th>
<th>8E8</th>
<th>8E9</th>
<th>8EA</th>
<th>8EB</th>
<th>8EC</th>
<th>8ED</th>
<th>8EE</th>
<th>8EF</th>
<th>8F0</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>
<tr>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td>halfword</td>
<td></td>
</tr>
<tr>
<td>word</td>
<td>word</td>
<td>doubleword</td>
<td>word</td>
<td>doubleword</td>
<td>word</td>
<td>doubleword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

- Instructions may reference single bytes, groups as shown, or strings of bytes of (almost) any length
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>General Register 12</td>
<td>General Register 13</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
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

![Floating-point registers diagram]

- 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>Opcode</th>
<th>Regs</th>
<th>Addressing Halfword</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© IBM 2015 System z Assembler Language
Instruction Lengths

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

\[
\begin{align*}
00xxxxxx & \quad \text{2-byte instructions such as RR-type} \\
01xxxxxx & \quad \text{4-byte instructions such as RX-type} \\
10xxxxxx & \quad \text{4-byte instructions such as RS- and SI-type} \\
11xxxxxx & \quad \text{6-byte instructions such as SS-type}
\end{align*}
\]

- 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></td>
<td></td>
<td>Not available</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>
Interruptions

- The basic instruction cycle is modified to handle interruptions:

![Flowchart showing the process of handling interruptions]

- There are six *classes* of interruption:
  1. Restart (operator action)
  2. External (timer, clock comparator)
  3. Machine Check (processor malfunction)
  4. Input-Output (an I/O device has signaled a condition)
  5. Program (exception condition during program execution)
  6. Supervisor Call (program requests an Operating System service)
• The CPU saves the current ("old") PSW, loads a new PSW
  - Supervisor saves status information, processes the condition
  - Supervisor can return to interrupted program by loading old PSW
  - Applications are concerned almost entirely with Program interruptions

• Some “popular” Program Interruption Codes (IC):
  IC=1  Invalid Operation Code.
  IC=4  Access, Protection: program has referred to an area of memory to which access is not allowed.
  IC=6  Specification Error: can be caused by many conditions.
  IC=7  Data Exception: invalid packed decimal data, or by floating-point conditions described in Chapter IX.
  IC=8  Fixed-Point Overflow: fixed-point binary result too large.
  IC=9  Fixed-Point Divide Exception: quotient would be too big, or a divisor is zero.
  IC=A  Decimal Overflow: packed decimal result too large.
  IC=B  Decimal Divide: packed decimal quotient too large, or a divisor is zero.
  IC=C  Hexadecimal floating-point exponent overflow: result too large.
  IC=D  Hexadecimal floating-point exponent underflow: result too small.
• System z instructions create many operand addresses using *base-displacement addressing*

```
| 4 bits | 12 bits |
+--------+---------+
| base digit | displacement |
+------------+------------+
```

- The base register specification digit ("base digit") specifies one of general registers 1-15, the base register containing the base address or base
- The displacement is an unsigned 12-bit integer

**Operand addresses:**

1. Copy displacement to internal Effective Address Register ("EAR")
2a. If the base digit b is not zero, add c(Rb); ignore carries
2b. If the base digit b is zero, do nothing

**The result in the EAR is the Effective Address.**
- Example with 32-bit values: an addressing halfword contains X’B2D5’, and c(R11) = X’C73E90AF’. Then, in the EAR:

  Step 1: 000002D5 displacement
  Step 2a: +C73E90AF base
  C73E9384 Effective Address
Indexing and Virtual Addresses

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>r</th>
<th>x</th>
<th>b</th>
<th>displacement</th>
</tr>
</thead>
</table>

- Indexed Effective Address calculation adds two more steps:

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

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

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

- System z supports *Dynamic Address Translation* (“DAT”)
  - DAT translates application addresses into “real“ addresses
  - Helps Operating System better manage “actual” memory
    - Invisible to your program.
Chapter III. Assembler Language Programs

This chapter describes fundamental concepts of Assembler Language programming.

- Section 6 provides an overview of assembling, linking and loading for execution; conventions for preparing Assembler Language programs; and some helpful macro instructions that perform simple I/O and conversion operations.

- Section 7 discusses key concepts relating to symbols and “variables”.

- Section 8 investigates the elements of expression evaluation, and the basic Assembler Language operand formats used by instructions.

- Section 9 introduces typical instructions and how to write Assembler Language statements for them.

- Section 10 shows how the Assembler calculates displacements and assigns base register values in Addressing Halfwords, and introduces the important USING and DROP assembler instructions.
• The Assembler helps you prepare instructions for execution on System z
• Gives you maximum control over selection and sequencing of specific instructions
• Assembler Language itself is much simpler than other programming languages
• Main difficulties are
  - Learning an appropriate set of machine instructions for your applications
  - Learning all the auxiliary tools and programs needed to build and use Assembler Language programs
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:

```
1  10  20  30  40  50  60  70  80
...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)
• 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 C'John R. Ehrman' Define constant with name
      END Test    Last statement
/*
```

- “Job Control” statements (lines 1-3, 12) vary from system to system
Six macro instructions are used extensively throughout the text:

**PRINTOUT**
Display contents of registers and named areas of memory.

**READCARD**
Read an 80-character record into memory.

**PRINTLIN**
Send a string of characters to a printer.

**DUMPOUT**
Display contents of memory in hexadecimal and character formats.

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

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

Equivalent facilities may be available at your location.
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
A constant value held by the Assembler in a two’s complement 32-bit word

- **Decimal**: an unsigned string of decimal digits with value between 0 and \(2^{31} - 1\) (2147483647)

- **Hexadecimal**: a string of hexadecimal digits enclosed in \(X^{'}\ldots^{'}\) with value between \(X^{'}0^{'}\) and \(X^{'}FFFFFFFF^{'}\) (2^32 − 1, 4294967295)

- **Binary**: a string of binary digits enclosed in \(B^{'}\ldots^{'}\) with value between \(B^{'}0^{'}\) and \(B^{'}11111111111111111111111111111111^{'}\) (2^32 − 1, 4294967295)

- **Character**: a string of 1-4 characters enclosed in \(C^{'}\ldots^{'}\), 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>.</td>
<td>4B</td>
<td>a</td>
<td>81</td>
<td>b</td>
<td>82</td>
</tr>
<tr>
<td>c</td>
<td>83</td>
<td>d</td>
<td>84</td>
<td>e</td>
<td>85</td>
<td>f</td>
<td>86</td>
</tr>
<tr>
<td>g</td>
<td>87</td>
<td>h</td>
<td>88</td>
<td>i</td>
<td>89</td>
<td>j</td>
<td>91</td>
</tr>
<tr>
<td>k</td>
<td>92</td>
<td>l</td>
<td>93</td>
<td>m</td>
<td>94</td>
<td>n</td>
<td>95</td>
</tr>
<tr>
<td>o</td>
<td>96</td>
<td>p</td>
<td>97</td>
<td>q</td>
<td>98</td>
<td>r</td>
<td>99</td>
</tr>
<tr>
<td>s</td>
<td>A2</td>
<td>t</td>
<td>A3</td>
<td>u</td>
<td>A4</td>
<td>v</td>
<td>A5</td>
</tr>
<tr>
<td>w</td>
<td>A6</td>
<td>x</td>
<td>A7</td>
<td>y</td>
<td>A8</td>
<td>z</td>
<td>A9</td>
</tr>
<tr>
<td>A</td>
<td>C1</td>
<td>B</td>
<td>C2</td>
<td>C</td>
<td>C3</td>
<td>D</td>
<td>C4</td>
</tr>
<tr>
<td>E</td>
<td>C5</td>
<td>F</td>
<td>C6</td>
<td>G</td>
<td>C7</td>
<td>H</td>
<td>C8</td>
</tr>
<tr>
<td>I</td>
<td>C9</td>
<td>J</td>
<td>D1</td>
<td>K</td>
<td>D2</td>
<td>L</td>
<td>D3</td>
</tr>
<tr>
<td>M</td>
<td>D4</td>
<td>N</td>
<td>D5</td>
<td>O</td>
<td>D6</td>
<td>P</td>
<td>D7</td>
</tr>
<tr>
<td>Q</td>
<td>D8</td>
<td>R</td>
<td>D9</td>
<td>S</td>
<td>E2</td>
<td>T</td>
<td>E3</td>
</tr>
<tr>
<td>U</td>
<td>E4</td>
<td>V</td>
<td>E5</td>
<td>W</td>
<td>E6</td>
<td>X</td>
<td>E7</td>
</tr>
<tr>
<td>Y</td>
<td>E8</td>
<td>Z</td>
<td>E9</td>
<td>0</td>
<td>F0</td>
<td>1</td>
<td>F1</td>
</tr>
<tr>
<td>2</td>
<td>F2</td>
<td>3</td>
<td>F3</td>
<td>4</td>
<td>F4</td>
<td>5</td>
<td>F5</td>
</tr>
<tr>
<td>6</td>
<td>F6</td>
<td>7</td>
<td>F7</td>
<td>8</td>
<td>F8</td>
<td>9</td>
<td>F9</td>
</tr>
</tbody>
</table>
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 Assembler uses a *Location Counter* (LC) to keep track of assembly-time positions in your program
  - Current LC value is represented by an asterisk (*)
  - As instructions and data are generated, the Assembler adds the length of the generated data to form the location of the next item
    • Locations of instructions always rounded to an even location

• Important distinction:
  1. *Locations* refer to positions in the Assembler’s model of your program
  2. *Addresses* refer to positions in memory at execution time
Values are assigned to symbols in two ways:

1. Name-field symbols usually take the current value of the Location Counter (before adding the length of generated data)

   ```
   MyProgl  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 ’C1’
   ```

3. The length of the generated data is usually assigned as the symbol’s length attribute
Symbols in high-level languages (usually called “variables”) have execution-time values

\[ X \leftarrow 22./7. \; ; \quad \text{/* Set } X \text{ to an approximation to } \pi \text{ */} \]

Symbols in Assembler Language are used only at assembly time; they have no execution-time value (are NOT “variables”)

- Used as names of places in a program that may contain execution-time values
- Symbol values simply help to organize the program
• Typical machine instruction statement format:

```
symbol operation operand1,operand2,... remarks
(optinal) (required) ← 0 to many → (optional)
```

• Assembler Language operands are formed from expressions
  - Expressions are formed from terms and operators

• Operators are +, −, *, /

• Terms take several forms... (slide 18)
A basic expression element is a *term*

- A self-defining term (always absolute)
- A symbol (absolute or relocatable)
- A Location Counter reference * (always relocatable)
- A Literal (always relocatable)
- A symbol attribute reference (always absolute)
  - Length (L’symbol)
  - Integer (I’symbol)
  - Scale (S’symbol)
Expressions

- An expression is an arithmetic combination of terms and operators
  \[ 7 + 4 \times C' \times N/L' \text{Item Size*Count} \]
- A parenthesized expression is treated as a term
  \[ (A+2)*(X'4780'\text{JJ}) \quad (7)+(6-2) \]
  - Parenthesized sub-expressions are evaluated first
- Unary (prefix) + and - are allowed
1. Each term is evaluated to 32 bits, and its relocatability is noted
2. Inner parenthesized sub-expressions evaluated first (from inside to out)
3. At the same level, do multiplication and division before addition and subtraction
   
   So, 2+5*3─6 -> (2+(5*3)─6), not ((2+5)*3)─6

4. No relocatable terms allowed in multiplication or division
5. For same-priority operations, evaluate from left to right
   
   So, 5*2/4 -> (5*2)/4, 5/2*4 -> (5/2)*4

6. Multiplication retains low-order 32 bits of 64-bit product
7. Division discards any remainder
   • Division by zero is allowed; result is zero (!)

8. Evaluation result is a 32-bit two’s complement value

9. Relocatability attribute of expression determined from relocatability of terms:
   a. Pairs of terms with same attribute and opposite signs have no effect (they “cancel”); if all are paired, the expression is absolute
   b. One remaining unpaired term sets the attribute of the expression; + means simply relocatable, – means complexly relocatable
   c. More than one unpaired term means the expression is complexly relocatable (a rare occurrence)
Machine Instruction Statement Operand Formats

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

\[
\begin{align*}
\text{expr} & \quad \text{expr}_1(\text{expr}_2) & \quad \text{expr}_1(\text{expr}_2,\text{expr}_3) \\
7 & \quad 8(7) & \quad 22(22,22) \\
8*N+4 & \quad A(B) & \quad (A)((B),(C)) \\
(91) & \quad (91)(15) & \quad (91)(,15)
\end{align*}
\]

- In the second and third forms, adjacent parentheses do not imply multiplication!
- In the third form, \(\text{expr}_2\) can be omitted if it is zero:

\[
\text{expr}_1(\text{,expr}_3) \quad \text{[The comma is still required!]}\]
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
    - $\text{expr}_1$ – absolute or relocatable
    - $\text{expr}_1(\text{expr}_2)$ – $\text{expr}_1$ absolute or relocatable, $\text{expr}_2$ absolute
    - $\text{expr}_1(\text{expr}_2,\text{expr}_3)$ – all three $\text{expr}$’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>Mnem</th>
<th>Instruction</th>
<th>Op</th>
<th>Mnem</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 $R_1, R_2$ – operands of “expr$_1$” form
• Assembler must generate machine language form of the instruction:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R(_1)</th>
<th>R(_2)</th>
</tr>
</thead>
</table>

• R\(_1\) and R\(_2\) designate first and second operand registers, \textit{not} general registers 1 and 2!

• Since LR opcode is \(X'18'\):

\[
\text{LR } 7,3 \quad \text{assembles to } X'1873'
\]

• Operands can be written as any expression with value \(0 \leq \text{value} \leq 15\)

\[
\text{LR } 3*4-5,1+1+1 \quad \text{also assembles to } X'1873'
\]

  - Assembly-time operands are \textit{expressions} with value “7” and “3”
  - Execution-time operands are \textit{contents} of general registers GR7 and GR3
These are some commonly used RX-type instructions:

<table>
<thead>
<tr>
<th>Op</th>
<th>Mnem</th>
<th>Instruction</th>
<th>Op</th>
<th>Mnem</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 $R_1$ (“expr$_1$” form)

Second operand field described as $S_2$ (“expr$_1$” form), as $S_2(X_2)$ (“expr$_1$(expr$_2$)” form), as $D_2(X_2,B_2)$, (“expr$_1$(expr$_2$,expr$_3$)” form), or as $D_2(,B_2)$ (“expr$_1$(,expr$_3$)” form)
Writing RX-Type Instructions

- Assembler must generate machine language form of the instruction:

<table>
<thead>
<tr>
<th>opcode</th>
<th>R_1</th>
<th>X_2</th>
<th>B_2</th>
<th>D_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>1</td>
<td>9</td>
<td>C</td>
<td>0C8</td>
</tr>
</tbody>
</table>

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

  L 1,200(9,12) will generate

  | 58 | 1 | 9 | C | 0C8 |

  L 1,200(,12) will generate

  | 58 | 1 | 0 | C | 0C8 |
Explicit and Implied Addresses

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

- Examples of explicit addresses:
  
<table>
<thead>
<tr>
<th>Code</th>
<th>IC</th>
<th>Displacement</th>
<th>$D_2$</th>
<th>$X_2$</th>
<th>$B_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>430A7468</td>
<td>IC</td>
<td>0,1128(10,7)</td>
<td>$D_2=1128$, $X_2=10$, $B_2=7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43007468</td>
<td>IC</td>
<td>0,1128(0,7)</td>
<td>$D_2=1128$, $X_2=0$, $B_2=7$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43070468</td>
<td>IC</td>
<td>0,1128(7,0)</td>
<td>$D_2=1128$, $X_2=7$, $B_2=0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- General forms of RX-instruction second operands:

<table>
<thead>
<tr>
<th></th>
<th>Explicit Address</th>
<th>Implied Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Indexed</td>
<td>$D_2(B_2)$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$D_2(X_2, B_2)$</td>
<td>$S_2(X_2)$</td>
</tr>
</tbody>
</table>
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>SLDL</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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

  | Instruction | Addressing | Remarks
  |-------------|------------|-------
  | 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) |

© IBM 2015 System z Assembler Language
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>

- \( D_1(B_1) \) (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>( D_1(B_1),I_2 )</td>
<td>( S_1,I_2 )</td>
</tr>
</tbody>
</table>

- Examples of SI-type instructions

  - MVI 0(6),C′∗’ Explicit \( D_1(B_1) \) address
  - CLI Buffer,C′0′ Implied \( S_1 \) address
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
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>
</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>
</table>

Two Length fields ("SS-2")

Addresses and lengths can both be specified explicitly or implicitly.

<table>
<thead>
<tr>
<th>SS-1 Form</th>
<th>Explicit Addresses</th>
<th>Implied Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Length</td>
<td>D₁(N₁,B₁),D₂(B₂)</td>
<td>S₁(N₁),S₂</td>
</tr>
<tr>
<td>Implied Length</td>
<td>D₁(B₁),D₂(B₂)</td>
<td>S₁,S₂</td>
</tr>
</tbody>
</table>

You write the length as N₁; the Assembler subtracts one to form L₁.

Some examples of SS-1 form instructions:

- MVC 0(80,4),40(9) Explicit length and addresses
- CLC Name(24),RecName Explicit length, implied addresses
- TR OutCh(,15),7(12) Implied length, explicit addresses
- XC Count,Count Implied length and addresses
For SS-2 form instructions, either or both operands may have explicit or implied addresses or lengths.

- This table shows some of the possible operand combinations:

<table>
<thead>
<tr>
<th>SS-2 Form</th>
<th>Explicit Addresses</th>
<th>Implied Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Lengths</td>
<td>$D_1(N_1,B_1), D_2(N_2,B_2)$</td>
<td>$S_1(N_1), S_2(N_2)$</td>
</tr>
<tr>
<td>Implied Lengths</td>
<td>$D_1,B_1, D_2,B_2$</td>
<td>$S_1, S_2$</td>
</tr>
</tbody>
</table>

Some examples of SS-2 form instructions:

- **PACK** 0(8,4),40(5,9)  Explicit lengths and addresses
- **ZAP** Sum(14),OldSum(4)  Explicit lengths, implied addresses
- **AP** Total(,15),Num(,12)  Implied lengths, explicit addresses
- **UNPK** String,Data  Implied lengths and addresses
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!
A common method for establishing execution-time addressability uses the RR-type BASR ("Branch and Save") instruction

\[ \text{BASR } R_1, R_2 \]

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

- The address in \( R_1 \) can be used as a base address
- \( R_1 \) can be used as a base register
Suppose we assemble this little program fragment, and that we know it will be loaded into memory at address \( X'5000' \)

```
5000  START  X'5000'  Starting location
5000  BASR  6,0  Establish base address
5002  BEGIN  L  2,N  Load contents of N into GR2
5006  A  2,ONE  Add contents of ONE
500A  ST  2,N  Store contents of GR2 into N

---twenty-two (X'16') additional bytes of instructions, data, etc.---
5024  N  DC  F'8'  Define Constant word integer 8
5028  ONE  DC  F'1'  Define Constant word integer 1
```

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

- At execution time, after the BASR is executed, \( c(R6)=X'00005002' \)
  - Since the L instruction wants to refer to the word at \( X'5024' \), its displacement from \( X'5002 \) is \( X'5024'-X'5002'=X'022' \)
  - So if we create an addressing halfword \( X'6022' \) for the L instruction, we know that when the L is executed it will refer to the correct address

- We can continue this way...
• Suppose this fragment is to be loaded into memory at address \(X'84E8'\)

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembled Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>84E8</td>
<td>OD60</td>
</tr>
<tr>
<td>84EA</td>
<td>58206022</td>
</tr>
<tr>
<td>84EE</td>
<td>5A206026</td>
</tr>
<tr>
<td>84F2</td>
<td>50206022</td>
</tr>
<tr>
<td>850C</td>
<td>00000008</td>
</tr>
<tr>
<td>8510</td>
<td>00000001</td>
</tr>
</tbody>
</table>

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:
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>BASR</td>
<td>6,0</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,X'022'(0,6)</td>
</tr>
<tr>
<td>0006</td>
<td>A</td>
<td>2,X'026'(0,6)</td>
<td></td>
</tr>
<tr>
<td>000A</td>
<td>ST</td>
<td>2,X'022'(0,6)</td>
<td></td>
</tr>
</tbody>
</table>

- 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>BASR</td>
<td>6,0</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>BEGIN</td>
<td>L</td>
<td>2,N-BEGIN(0,6) (N-BEGIN = X'022')</td>
</tr>
<tr>
<td>0006</td>
<td>A</td>
<td>2,ONE-BEGIN(0,6) (ONE-BEGIN = X'026')</td>
<td></td>
</tr>
<tr>
<td>000A</td>
<td>ST</td>
<td>2,N-BEGIN(0,6) (N-BEGIN = X'022')</td>
<td></td>
</tr>
</tbody>
</table>

- 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:

  ```
  BASR 6,0                        GR6 will hold execution address of BEGIN
  USING BEGIN, 6                  Tell Assembler C(GR6)=address of BEGIN
  BEGIN L 2,N                    Load c(N) into GR2
  A 2,ONE                        Add c(ONE) to GR2
  ST 2,N                         Store sum at N

  * ──── 22 bytes ────
  N DC F'8'
  ONE DC F'1'
  ```

- And the Assembler does the hard work!
Location Counter Reference

- The Assembler builds an accurate model of your program using the *Location Counter* (LC)
  - The position of each piece of object code (or reserved space) is given a *location* during assembly
  - The relative positions of all items in each major segment of a program is fixed at assembly time

- “*” as a term has the value of the Location Counter, so we can use a *Location Counter Reference*
  
  **BASR 6,0** Establish base register
  **USING *,6** Tell Assembler base location is “here”

- No need to define a symbol just for use in the USING statement; specifying a symbol on the instruction following a BASR is (a) inconvenient, (b) unnecessary, and (c) sometimes a poor idea
Suppose we make a mistake in the L instruction named **BEGIN**:

<table>
<thead>
<tr>
<th>Location</th>
<th>Object Code</th>
<th>Statement</th>
<th>Location</th>
<th>Object Code</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0D60</td>
<td>BASR 6,0</td>
<td>0002</td>
<td>58606022</td>
<td>BEGIN L 6,N ←Wrong register! (6, not 2)</td>
</tr>
<tr>
<td>0006</td>
<td>5A206026</td>
<td>A 2,ONE</td>
<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>
<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’00005002’
- When the L instruction is executed, c(GR6) = X’00000008’
- When the A instruction is executed, its Effective Address is X’00000002E’ (not X’00005028’!)
- Worse: the ST will try to store into memory at address X’00000002A’, probably causing a memory protection exception

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

  \[
  \text{displacement} = (\text{implied	extunderscore address value}) - (\text{base	extunderscore location value})
  \]

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

• If successful, the Assembler has \textit{resolved} the implied address. If not, the implied address is \textit{not addressable}

• The Assembler does at assembly time the reverse of what the CPU does at execution time:

  Assembly: \text{displacement} = (\text{operand	extunderscore location}) - (\text{base	extunderscore location})
  
  Execution: \text{(operand address)} = \text{displacement} + (\text{base address})
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>
<tr>
<td>0006</td>
<td>A 2,ONE</td>
</tr>
</tbody>
</table>

- 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’000000028’–X’000000026’ = X’026’ with addressing halfword X’6026’
  - With register 7: X’000000028’–X’000000006’ = 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
  
  \[
  \text{DROP register}
  \]
  
  - In the previous examples, if we write
    
    \[
    \text{DROP 6}
    \]
    
    then the USING Table looks like this:

    | basereg | base_location | RA |
    |---------|---------------|----|
    | empty   |               |    |
    | 7       | 00000006      | 01 |

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

    | basereg | base_location | RA |
    |---------|---------------|----|
    | empty   |               |    |
    | empty   |               |    |
Many conditions can cause addressability problems

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

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

2. If the USING Table is empty, implied addresses can’t be resolved
   - Except for resolutions of absolute implied addresses with register 0; see slide 48

3. Symbols in other sections have different Relocatability Attributes
   - Techniques used to refer to them are described in Chapter X

4. Complexly relocatable operands (rare!) are never addressable
The Assembler has an implied USING Table entry for register 0:

<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>etc.</td>
<td></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:

\[
\begin{align*}
\text{LA 7,100} & & 100 = '064', \text{ so addressing halfword = '0064'} \\
\text{LA 7,4000} & & 4000 = 'FA0', \text{ so addressing halfword = '0FA0'}
\end{align*}
\]

You can specify an absolute base_location in a USING statement:

\[
\begin{align*}
\text{LA 9,400} & \quad \text{Base address} = 400 = '190' \\
\text{USING 400,9} & \quad \text{Base address} = 400 = '190' \\
\text{LA 3,1000} & \quad 1000 = '3E8'
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>DC</th>
<th>F'8'</th>
<th>Word binary integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>C'/'</td>
<td>Character constant</td>
</tr>
<tr>
<td>DC</td>
<td>X'61'</td>
<td>Hexadecimal constant</td>
</tr>
<tr>
<td>DC</td>
<td>B'01100001'</td>
<td>Binary constant</td>
</tr>
</tbody>
</table>

  - Note that the last three constants use the same nominal value representation as the corresponding self-defining terms.
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
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:
  \[
  Ln \quad \text{or} \quad L(\text{expr})
  \]
- Examples:

  \[\begin{array}{ccc}
  A & DC & FL3'8'
  \end{array}\]
  Generates X'000008' at current location

  \[\begin{array}{ccc}
  B & DC & FL(2*4-5)'8'
  \end{array}\]
  Generates X'000008' at current location

  \[\begin{array}{ccc}
  C & DC & F'8'
  \end{array}\]
  Generates X'00000008' at next word boundary

  \[\begin{array}{ccc}
  D & DC & FL4'8'
  \end{array}\]
  Generates X'00000008' at current location

- Symbols $A$, $B$, $D$ are given values of the LC where the constants are generated; symbol $C$ is given the LC value after bytes are skipped for alignment
You can generate copies of a constant in several ways:

- Multiple operands
  
  ```assembly
  DC F'8', F'8', F'8'
  ```

- Multiple statements:
  
  ```assembly
  DC F'8'
  DC F'8'
  DC F'8'
  DC F'8'
  ```

- Duplication factors:
  
  ```assembly
  DC 3F'8'
  ```

- Duplication factors can be used on each operand:
  
  ```assembly
  DC 2F'8', 2F'29', F'71', 3F'2' 8 word constants
  ```
Multiple Nominal Values

- Almost all constant types accept multiple nominal values, separated by commas.
  
  \[
  \begin{align*}
  \text{DC F'} & 1,2,3,4,5' & \text{Five word constants} \\
  \text{DC X'} & A,B,C,D,7' & \text{Five one-byte constants} \\
  \text{DC B'} & 001,010,011' & \text{Three one-byte constants}
  \end{align*}
  \]

- Many constant types accept embedded spaces for readability:
  
  \[
  \begin{align*}
  \text{DC F'} & 1 000 000 000' & \text{Easier than counting adjacent zeros} \\
  \text{DC X'} & 1 234 567 89A' & \text{Five-byte constant}
  \end{align*}
  \]

- Character constants are the exception: a comma or a space is part of the nominal value.
  
  \[
  \begin{align*}
  \text{DC C'} & 1,2,3,4,5' & \text{One nine-byte constant} \\
  \text{DC C'} & 1 2 3 4 5' & \text{One nine-byte constant}
  \end{align*}
  \]
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
  
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Argument</th>
<th>LA of Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>LR 7,3</td>
<td>2</td>
</tr>
<tr>
<td>BEGIN</td>
<td>L 2,N</td>
<td>4</td>
</tr>
</tbody>
</table>

- LA of symbols naming DC statements is the length of the _first_ generated constant, _ignoring_ duplication factors
  
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Argument</th>
<th>LA of Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied DC</td>
<td>F'8'</td>
<td>4</td>
</tr>
<tr>
<td>Explicit DC</td>
<td>XL7'ABC'</td>
<td>7</td>
</tr>
<tr>
<td>Multiple DC</td>
<td>3F'8'</td>
<td>4</td>
</tr>
<tr>
<td>List DC</td>
<td>F'1,2,3'</td>
<td>4</td>
</tr>
<tr>
<td>OddOnes DC</td>
<td>B'1',F'2',X'345'</td>
<td>1</td>
</tr>
</tbody>
</table>

- For almost all EQU statements, the Assembler assigns LA 1:
  
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Argument</th>
<th>LA of Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>R7</td>
<td>Equ 7</td>
<td>1</td>
</tr>
</tbody>
</table>
Decimal Exponents

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

  - BillionA DC   F’1000000000’ The hard way
  - BillionB DC   F’1 000 000 000’ An easier way
  - BillionC DC   F’1E9’ The easiest way

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

  - HundredA DC   F’1E2’ Generates X’00000064’
  - HundredB DC   F’1000E−1’ Generates X’00000064’

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

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

- Decimal exponents and exponent modifiers are often used in floating-point constants
Section 12 provides more detail about seven basic constant types:

- **F** Two’s complement binary integers, normally word length
- **H** Two’s complement binary integers, normally halfword length
- **A** Address- or expression-valued, normally word length
- **Y** Expression-valued, normally halfword length
- **C** Character-valued constant, length 1 to 256 bytes
- **B** Bit-valued constant, length 1 to 256 bytes
- **X** Hexadecimal-valued constant, length 1 to 256 bytes

- Literals provide a powerful way to define and address constants
F-Type and H-Type Constants

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

\[
\begin{align*}
\text{DC } & \text{ F' } -10' \quad \text{Generates } X'\text{FFFFFFF6}', \text{ word aligned} \\
\text{DC } & \text{ H' } -10' \quad \text{Generates } X'\text{FFF6}', \text{ halfword aligned}
\end{align*}
\]

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

\[
\begin{align*}
\text{DC } & \text{ FL5' } -10' \quad \text{Generates } X'\text{FFFFFFF6}', \text{ unaligned} \\
\text{DC } & \text{ HL5' } -10' \quad \text{Generates } X'\text{FFFFFFF6}', \text{ unaligned}
\end{align*}
\]

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

\[
\begin{align*}
\text{DC } & \text{ F' U2147483648' } \quad \text{Generates } X'\text{80000000}', \text{ word aligned } (2^{31}) \\
\text{DC } & \text{ H' U65535' } \quad \text{Generates } X'\text{FFFF}', \text{ halfword aligned } (2^{16}-1) \\
\text{DC } & \text{ F' U4294967295' } \quad \text{Generates } X'\text{FFFFFFF}', \text{ word aligned } (2^{32}-1) \\
\text{DC } & \text{ H' 1, U2' } \quad \text{Generates } X'\text{00010002}', \text{ Mixed forms}
\end{align*}
\]
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:

<table>
<thead>
<tr>
<th>R7</th>
<th>Equ 7</th>
<th>Define a symbol value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr1</td>
<td>DC A(C’A’+48)</td>
<td>Generates X’000000F1’, word aligned</td>
</tr>
<tr>
<td></td>
<td>DC A(R7)</td>
<td>Generates X’00000007’, word aligned</td>
</tr>
<tr>
<td></td>
<td>DC A(Expr1)</td>
<td>Will contain execution–time address of Expr1</td>
</tr>
<tr>
<td></td>
<td>DC AL1(*–Expr1)</td>
<td>Will generate X’0C’, unaligned</td>
</tr>
<tr>
<td>Here</td>
<td>DC A(*+64)</td>
<td>Will contain execution–time address of Here+64</td>
</tr>
</tbody>
</table>

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

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

- Early (and very small) machines used relocatable 16-bit address constants

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

  A  DC  C’12345’  Generates X’C1C2C3C4’;  LA of A = 5
  B  DC  X’123456’ Generates X’123456’;  LA of B = 3
  C  DC  2B’10100101’ Generates X’A5A5’;  LA of C = 1

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

  DC  C’’’’’  Generates X’7D’
  DC  C’&&'&&  Generates X’5050’
The space allocated for a constant is defined either by default or by a length modifier.

If the constant is too small for the space, it must be *padded*; if the constant is too large for the space, it must be *truncated*.

The Assembler’s actions in such cases depends on the constant type:

<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>
• A literal is a type of symbol that references and defines a constant
• Written as an equal sign followed by a DC operand
  \[ =F'8' =H'22' =4X'40' =CL2' $' =A(X'40',C'$) \]
• Some limitations and restrictions:
  - Multiple operands are not allowed (but multiple values are OK)
  - Duplication factors are allowed, but may not be zero
  - Literals are not allowed as operands of address constants
• The Assembler tries to diagnose instructions that can directly modify a literal
  \[
  \text{L } 2,=F'8' \quad \text{Valid reference} \\
  \text{ST } 2,=F'8' \quad \text{Invalid; tries to modify the literal}
  \]
• A literal is more likely to be a “constant” constant
The Assembler collects literals internally as they are referenced; they must be assembled somewhere into your program.

The LTORG (“Literal Origin”) instruction lets you specify where the collected literals should be placed:
- It’s important that all literals are addressable!

Literals are placed in the program in order of decreasing alignment requirement:
- Assembler’s internal collection is emptied

Subsequent literal references start a new collection:
- A subsequent LTORG will generate the new ones
  - The same literal can appear more than once; they are treated as distinct symbols

At the END statement, all remaining literals are generated.
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'1E15'          | X'00038D7EA4C68000' |
    | DC | AD(C' ABC')       | X'0000000000C1C2C3' |

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

| DC | C' ABC'           | X'C1C2C3'         | EBCDIC (TRANSLATE-able) |
| DC | CE' ABC'          | X'C1C2C3'         | EBCDIC always           |
| DC | CA' ABC'          | X'414243'         | ASCII always            |
| DC | CU' ABC'          | X'004100420043'   | Unicode UTF-16          |
Section 13 shows ways to define and organize data and work areas
- The DS ("Define Storage") instruction is similar to DC, but generates no object code
- The EQU ("Equate") instruction lets you assign values to symbols, or define similarities of one symbol to another
- The ORG ("Set Origin") instruction lets you adjust the position of the Location Counter

With combinations of these instructions, you can define data structures that greatly simplify many programming tasks
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

  \[
  \begin{align*}
  &\text{DS } F & \text{Both statements allocate 4 bytes of ...} \\
  &\text{DS } F'8' & \text{uninitialized space on a word boundary}
  \end{align*}
  \]

- Multiple operands and values are allowed

  \[
  \begin{align*}
  &\text{DS } F'1', X' \text{ABC}', C' \text{ABC}' & \text{Allocates 9 bytes, word aligned} \\
  &\text{DS } H'2,4,6,8' & \text{Allocates 8 bytes, halfword aligned}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  &\text{DS } F,X & \text{Allocate 5 bytes, word aligned} \\
  &\text{DS } F & \text{Skip 3 bytes for word alignment}
  \end{align*}
  \]

- Length Attributes of names are derived from first operand

  \[
  \begin{align*}
  &\text{Area1 } \text{DS 80C} & \text{Allocate 80 bytes; LA of Area1 = 1} \\
  &\text{Area2 } \text{DS CL80} & \text{Allocate 80 bytes; LA of Area2 = 80}
  \end{align*}
  \]

  - Both statements allocate 80 bytes, unaligned
Zero Duplication Factor

- Zero duplication factor can be used with DS and DC
  - Boundary alignment and symbol attributes done “as usual”
    
    DC OF'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 OF'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
Extended EQU syntax supports up to 5 operands (the last two are used for conditional assembly and macros)

\[
symbol \text{ EQU } \text{value, length, type[, program-attribute, assembler-attribute]}\]

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

Extended syntax is usually used with just the first two operands

We can rewrite the Phone Number example to use extended syntax:

```
PhoneNum DS CL10 Space for entire number
AreaCode Equ PhoneNum,3,C’C’ Overlay AreaCode
Prefix Equ AreaCode+3,3,C’C’ Overlay Prefix
Local_No Equ Prefix+3,4,C’C’ Overlay Local_No
```
The ORG Assembler Instruction

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

\[
\text{ORG } \text{relocatable\_expression}
\]

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

\[
\begin{align*}
\text{PhoneNum DS CL10} & \quad \text{Space for entire number} \\
\text{ORG PhoneNum} & \quad \text{Reposition Location Counter} \\
\text{AreaCode DS CL3} & \quad \text{Define AreaCode} \\
\text{Prefix DS CL3} & \quad \text{Define Prefix} \\
\text{Local\_No DS CL4} & \quad \text{Define Local\_No}
\end{align*}
\]

• ORG also supports an extended syntax:

\[
\begin{align*}
\text{ORG } \text{relocatable\_expression, boundary, offset} \\
\text{ORTH } *, 8, +6 & \quad \text{Set to 2 bytes before a doubleword boundary}
\end{align*}
\]

- The LC is first set to the expression value; then it is rounded up to the next power-of-two boundary; and finally the offset is added

• Exercise: Why +6? Why not ORG *, 8, -2?
Parameterization uses a small number of values to define and control constants, data areas, field lengths, offsets, etc.
- Many examples of parameterization were shown on previous slides

Suppose we must read, modify, and write 80-byte records

```
RecLen    Equ    80    Record length (for now)
InRec     DS     CL(RecLen) Area for input records
WorkRec   DS     CL(RecLen) Work area for records
OutRec    DS     CL(RecLen) Area for output records
```
- If the record length is changed, only the EQU statement needs updating

Suppose a table of 45 records must be maintained in storage

```
NRecs     Equ    45    Number of records in storage
RecNum    DC     Y(NRecs) Constant with number of records allowed
StorRecs  DS     (NRecs─1)CL(RecLen) Space for all but one records
LastRec   DS     CL(RecLen) Last record in storage
```
- Changing the number of records and the allocated space is a simple update

Parameterization simplifies *many* aspects of programming!
- It improves program readability and understandability
• Address constants usually refer to locations internal (or external) to a program

• They can also be used to generate tables of constants
  - Example: table of byte integers from 0 to 10:

    `IntTbl DC FL1'0,1,2,3,4,5,6,7,8,9,10' Generates 0,1,...9,10`
    or

    `IntTbl DC 11AL1(*-IntTbl) Generates 0,1,...9,10`

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

    `IntTbl2 DC FL1'10,9,8,7,6,5,4,3,2,1,0' Generates 10,9,...1,0`
    or

    `IntTbl2 DC 11AL1(IntTbl2-+10) Generates 10,9,...1,0`

• Very complex tables can be created using such techniques

• Exercise: Which `IntTbls` are easier to expand?
Slide 24
- Suppose the LC is already at a doubleword boundary: then the offset -2 might back up the LC over existing object code or data areas.

Slide 26
- The ones with AL1-type constants only need to modify the duplication factor, not the nominal value as in the FL1-type constants.
  - But for the example using \texttt{IntTbl2}, you'll need to change the $+10$ to one less than the duplication factor.
  - A better way:

\begin{verbatim}
NumInt2  EQU  11  Number of generated values
IntTbl2  DC  (NumInt2)AL1(IntTbl2-\ast\ast+NumInt2-1)
\end{verbatim}

- 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
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
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:

  \[
  \begin{align*}
  \text{L} & \quad 7, -F' -97' & \text{c(GR7) replaced by X' FFFFFFF9'} \\
  \text{ST} & \quad 7, \text{Num} & \text{c(NUM) replaced by c(GR7), GR7 unchanged}
  \end{align*}
  \]
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

\[
\begin{array}{ccc}
L & 1, A & ST & 1, B \\
L & 2, A+4 & & ST & 2, B+4 \\
L & 3, A+8 & & ST & 3, B+8 \\
\end{array}
\]

you can write

\[
\begin{array}{ccc}
LM & 1, 3, A & and & STM & 1, 3, B \\
\end{array}
\]

- The instruction format is

\[
\begin{array}{ccc}
LM (or STM) & R_1, R_3, D_2 (B_2) & (explicit address) \\
LM (or STM) & R_1, R_3, S_2 & (implied address) \\
\end{array}
\]

- The register contents are transmitted between GRs and successive words in memory, starting with the R_1 register and ending with the R_3 register.
  - If R_3 is smaller than R_1 then GRs R_1-15 are transmitted followed by GRs 0-R_3.
- These instructions are often used for “status preservation”
The RX-type Load Halfword (LH) and Store Halfword (STH) instructions transfer two bytes between memory and the rightmost 2 bytes of a GR:

- STH simply stores the 2 bytes at the Effective Address
- LH puts the 2 bytes in the right end of the GR, and sign-extends the leftmost through the rest of the GR

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

\[
\begin{align*}
\text{L} & 0,=F'65537' & c(\text{GR0}) = X'00010001' & +65537 = 2^{16}+1 \\
\text{STH} & 0,A & c(A) = X'0001' & \text{Lost a bit!} \\
\text{LH} & 1,A & c(\text{GR1}) = X'00000001' & \text{Lost significance!} \\
\text{L} & 0,=F'65535' & c(\text{GR0}) = X'0000FFFF' & +65535 = 2^{16}-1 \\
\text{STH} & 0,A & c(A) = X'FFFF' & \text{No lost bits, but wrong sign} \\
\text{LH} & 1,A & c(\text{GR1}) = X'FFFFFFFF' & (-1!) \text{Lost significance!}
\end{align*}
\]
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
 └─────────────────────────────┴─────────┘
```

- Two examples:

1. (1) `L 0,=F′−1′ c(GR1) = X′FFFFFFF′
   `IC 0,=C′A′ c(GR1) = X′FFFFFFC1′

2. (2) `IC 0,X Get 1st byte of c(X)
   `STC 0,Y+1 Store at 2nd byte of Y
   `IC 0,X+1 Get 2nd byte of c(X)
   `STC 0,Y Store byte at Y

```
X  DC  C′AB′
Y  DS  CL2  c(Y) becomes C′BA′
```
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.

![ICM (or STCM) R₁,M₃,D₂(B₂) (explicit address)](image)

- Bits positions of the M₃ “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</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M₃ = 0, or all inserted bytes are zero</td>
</tr>
<tr>
<td>1</td>
<td>Leftmost bit of first inserted byte = 1</td>
</tr>
<tr>
<td>2</td>
<td>Leftmost bit of first inserted byte = 0</td>
</tr>
</tbody>
</table>

- Example: suppose c(GR1) = 'X′ AABBCDD′

  ![ICM 1,B′0101′,=X′1122′](image)

  c(GR1) now = 'X′ AA11CC22′, CC=2

  ![STCM 1,B′1010′,Z](image)

  c(Z) now = 'X′ AACC′
Many instructions copy data among registers; some complement, extend, or test the operand; the $R_1$ and $R_2$ operands need not differ

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR 0,1</td>
<td>c(GR R1) ← c(GR R2)</td>
<td>Unchanged</td>
</tr>
<tr>
<td>LTR 1,1</td>
<td>c(GR R1) ← c(GR R2)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCR 2,1</td>
<td>c(GR R1) ← −c(GR R2)</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>LPR 3,1</td>
<td>c(GR R1) ←</td>
<td>c(GR R2)</td>
</tr>
<tr>
<td>LNR 3,1</td>
<td>c(GR R1) ← −</td>
<td>c(GR R2)</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
  - LR 0,1: c(R0) ← c(R1), CC unchanged
  - LTR 1,1: Test c(R1), set CC
  - LCR 2,1: c(R2) = −c(R1), set CC
  - LNR 3,1: c(R2) = −|c(R1)|, set CC
Load, Store, and Insert for 64-bit General Registers

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

![Diagram of 64-bit GRs]

- 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>(R_1)</th>
<th>(X_2)</th>
<th>(B_2)</th>
<th>(DL_2)</th>
<th>(DH_2)</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>(R_1)</td>
<td>(R_3)</td>
<td>(B_2)</td>
<td>(DL_2)</td>
<td>(DH_2)</td>
<td>opcode</td>
</tr>
</tbody>
</table>

  - Their 20-bit base-displacement format is described in Section 20
These instructions are similar to the 32-bit forms, but now have G in the mnemonic. (We sometimes use “GR” for 32-bit registers, and “GG” for 64-bit registers.) The CC settings are as shown on slide 8.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGR</td>
<td>c(GG R1) ← c(GG R2)</td>
<td>Not changed</td>
</tr>
<tr>
<td>LTGR</td>
<td>c(GG R1) ← c(GG R2)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCGR</td>
<td>c(GG R1) ← −c(GG R2)</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>LPGR</td>
<td>c(GG R1) ←</td>
<td>c(GG R2)</td>
</tr>
<tr>
<td>LNGR</td>
<td>c(GG R1) ← −</td>
<td>c(GG R2)</td>
</tr>
</tbody>
</table>

Examples:

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGR 7,3</td>
<td>c(GG7)=0, CC not set</td>
<td></td>
</tr>
<tr>
<td>LTGR 2,2</td>
<td>c(GG2)=1, CC=2</td>
<td></td>
</tr>
<tr>
<td>LNGR 1,3</td>
<td>c(GG1)=0, CC=0</td>
<td></td>
</tr>
<tr>
<td>LCGR 4,2</td>
<td>c(GG4)=-1, CC=1</td>
<td></td>
</tr>
<tr>
<td>LPGR 0,4</td>
<td>c(GG0)=+1, CC=2</td>
<td></td>
</tr>
<tr>
<td>LNGR 5,2</td>
<td>c(GG5)=-1, CC=1</td>
<td></td>
</tr>
</tbody>
</table>
Load and Test Instructions

- These instructions can replace an equivalent pair:

  \[
  \begin{align*}
  LT & \quad R_1, S_2 \\
  \text{replaces} & \\
  L & \quad R_1, S_2 \\
  LTR & \quad R_1, R_1 \\
  \text{or even} & \\
  ICM & \quad R_1, B'1111', S_2 \\
  \text{and,} & \\
  LTG & \quad R_1, S_2 \\
  \text{replaces} & \\
  LG & \quad R_1, S_2 \\
  LTGR & \quad R_1, R_1 \\
  \end{align*}
  \]

- 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 R₁
These instructions automatically sign-extend a 32-bit operand to 64 bits

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>CC Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGF</td>
<td>c(GG R₁) ← c(Word in memory)</td>
<td>Not changed</td>
</tr>
<tr>
<td>LGFR</td>
<td>c(GG R₁) ← c(GR R₂)</td>
<td>Not changed</td>
</tr>
<tr>
<td>LTGFR</td>
<td>c(GG R₁) ← c(GR R₂)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LCGFR</td>
<td>c(GG R₁) ← c(GR R₂)</td>
<td>0,1,2</td>
</tr>
<tr>
<td>LPGFR</td>
<td>c(GG R₁) ←</td>
<td>c(GR R₂)</td>
</tr>
<tr>
<td>LNGFR</td>
<td>c(GG R₁) ← -</td>
<td>c(GR R₂)</td>
</tr>
</tbody>
</table>

### Example

LCGFR 0,1 is equivalent to

\[ \text{LGFR 0,1} \]

\[ \text{LGCR 0,0} \]
Other General Register Load Instructions

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

1. Since both L (Load) and LR (Load Register) load a general register, they must be equivalent.

2. Since L (Load) and ST (Store) are complementary instructions, then for LR (Load Register) there must be a STR (Store Register) instruction.
   • There’s no STR; just use LR with reversed operands

3. Because programmers often define symbols like R0, R1, ... R15 to refer to the general registers, it’s easy to assume that a symbol like R1 always means GR1.
   • But R1 is just a name for a number! You could just as well write

   \[
   \text{R92 EQU 1} \quad \text{“R92” means Register 92 ??}
   \]
   \[
   \text{L R92,XYZ} \quad \text{You could write something like this...}
   \]
   \[
   \text{L 1,XYZ} \quad \text{and get the same result as writing this!}
   \]
Many instructions set the value of the PSW’s 2-bit Condition Code (CC)

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

“Branch On Condition” instructions let you select alternate execution paths
  - Basic forms are BC (RX-type) and BCR (RR-type)

Many other branch types extend these basic forms
• If the branch condition (defined on next slide) is *not* met:
  - Continue execution with the next sequential instruction

• If the branch condition *is* met:
  - For the BC (RX-type) instruction, the branch address is the Effective Address
  - For the BCR (RR-type) instruction, the branch address is the address in GR $R_2$
    - But if the $R_2$ digit is zero, no branch occurs and execution continues with the next sequential instruction

• The Instruction Address (IA) in the PSW is replaced by the branch address
  - So the next instruction to be fetched is at the branch address
The branch condition is determined by testing a bit in the M\textsubscript{1} mask field of the instruction:

```
07 M1 R2
47 M1 X2 B2 D2
```

The current value of the CC selects a bit in the M\textsubscript{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 position</th>
<th>Mask bit position</th>
<th>Mask bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**BCR 9,4** \[M_1 = \text{B}'1001'\] **Branch if CC = 0,3**

**BC 7,4(8,2)** \[M_1 = \text{B}'0111'\] **Branch if CC = 1,2,3**
1. Branch to XX if the CC is zero.
   \[ BC \quad 8,XX \quad M_1 = 01000 \]

2. Branch to XX if the CC is not 0.
   \[ BC \quad 7,XX \quad M_1 = 00111 \]

3. Always branch to the instruction whose address is contained in GR14.
   \[ BCR \quad 15,14 \quad M_1 = 11111 \]
   or
   \[ BC \quad 15,0(0,14) \quad M_1 = 11111 \]
   - When all mask bits are 1, the CC value must match a 1-bit in the mask; this is called an \textit{unconditional branch}\n
4. Branch to XX if the CC is 1 or 3.
   \[ BC \quad 5,XX \quad M_1 = 00110 \]
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:
  
  \[
  \begin{align*}
  & BC \quad 0, x \\
  & BCR \quad 0, \text{any}
  \end{align*}
  \]

- The Assembler provides special “extended mnemonics” for them:
  
  \[
  \begin{align*}
  & NOP \quad S_2 \quad \text{is equivalent to} \quad BC \quad 0, S_2 \\
  & NOPR \quad R_2 \quad \text{is equivalent to} \quad BCR \quad 0, R_2
  \end{align*}
  \]

- Some no-operation instructions have special side-effects:
  
  \[
  \begin{align*}
  & BCR \quad 15, 0 \quad (\text{“branch always nowhere”})
  \end{align*}
  \]
  
  causes internal overlaps of fetch/decode/execute phases to slow
  - Sometimes helps with problem diagnosis or potential memory conflicts
Conditional No-Operation

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

\[
\text{CNOP boundary, width}
\]
- \textit{width} is either 4, 8, or 16
- \textit{boundary} is an even number such that \textit{boundary} < \textit{width}

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

<table>
<thead>
<tr>
<th>halfword1</th>
<th>halfword2</th>
<th>halfword3</th>
<th>halfword4</th>
<th>halfword5</th>
<th>halfword6</th>
<th>halfword7</th>
<th>halfword8</th>
</tr>
</thead>
<tbody>
<tr>
<td>word1</td>
<td>word2</td>
<td>word3</td>
<td>word4</td>
<td>word5</td>
<td>word6</td>
<td>word7</td>
<td>word8</td>
</tr>
<tr>
<td></td>
<td>doubleword1</td>
<td></td>
<td></td>
<td></td>
<td>doubleword2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,4</td>
<td>2,4</td>
<td>0,4</td>
<td>2,4</td>
<td>0,4</td>
<td>2,4</td>
<td>0,4</td>
<td>2,4</td>
</tr>
<tr>
<td>0,8</td>
<td>2,8</td>
<td>4,8</td>
<td>6,8</td>
<td>0,8</td>
<td>2,8</td>
<td>4,8</td>
<td>6,8</td>
</tr>
<tr>
<td>0,16</td>
<td>2,16</td>
<td>4,16</td>
<td>6,16</td>
<td>8,16</td>
<td>10,16</td>
<td>12,16</td>
<td>14,16</td>
</tr>
</tbody>
</table>

Chap. V, Sec. 15.5 System z Assembler Language ©IBM 2015
• 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>BNM</td>
<td>BNMR</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>BO</td>
<td>BOR</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>
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>SH</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>SR</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>AG</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>L 1,=F’2147483647’</th>
<th>2₃¹−1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1,=F’1’</td>
<td></td>
</tr>
<tr>
<td>L 2,=F’2147483647’</td>
<td>2₃¹−1</td>
</tr>
<tr>
<td>S 2,=F’1’</td>
<td></td>
</tr>
<tr>
<td>L 3,=F’–2147483648’</td>
<td>–2₃¹</td>
</tr>
<tr>
<td>S 3,=F’1’</td>
<td></td>
</tr>
<tr>
<td>LG 4,=X’70000000 00000000’</td>
<td>CC=3 (overflow)</td>
</tr>
<tr>
<td>AGR 4,4</td>
<td></td>
</tr>
<tr>
<td>LG 5,=X’50000000 00000000’</td>
<td>CC=3 (overflow)</td>
</tr>
<tr>
<td>SG 5,=X’60000000 00000000’</td>
<td>CC=1 (negative)</td>
</tr>
</tbody>
</table>
Signed-Arithmetic Compare Instructions

- Comparisons with 32-bit operands

<table>
<thead>
<tr>
<th>CH, CHY</th>
<th>Compare c(GR R1) to halfword in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, CY</td>
<td>Compare c(GR R1) to word in memory</td>
</tr>
<tr>
<td>CR</td>
<td>Compare c(GR R1) to word in c(GR R2)</td>
</tr>
</tbody>
</table>

- Comparisons with 64-bit operands

<table>
<thead>
<tr>
<th>CG</th>
<th>Compare c(GG R1) to doubleword in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGR</td>
<td>Compare c(GG R1) to doubleword in c(GG R2)</td>
</tr>
</tbody>
</table>

- Condition Code 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>

- The CC cannot be set to 3 as a result of a compare instruction
Logical arithmetic produces bitwise identical results as the equivalent arithmetic operation; only the CC settings are different.

- Instructions with 32- and 64-bit operands

<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(GR R₂)</td>
<td>SLR</td>
<td>Subtract word from c(GR R₂)</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(GG R₂)</td>
<td>SLGR</td>
<td>Subtract doubleword from c(GG R₂)</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(GR R₁) = c(GR R₁) ± c(GR R₂)</td>
<td>0: Zero result, no carry (Note)</td>
</tr>
<tr>
<td>c(GR R₁) = c(GR R₁) ± c(word in memory)</td>
<td>1: Nonzero result, no carry</td>
</tr>
<tr>
<td>c(GG R₁) = c(GG R₁) ± c(GG R₂)</td>
<td>2: Zero result, carry</td>
</tr>
<tr>
<td>c(GG R₁) = c(GG R₁) ± c(doubleword in memory)</td>
<td>3: Nonzero result, carry</td>
</tr>
</tbody>
</table>

**Note:** CC0 cannot occur for logical subtraction
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>ALC</td>
<td>Add word from memory</td>
<td>SLB</td>
<td>Subtract word from memory</td>
</tr>
<tr>
<td>ACLR</td>
<td>Add word from c(GR R2)</td>
<td>SLBR</td>
<td>Subtract word from c(GR R2)</td>
</tr>
<tr>
<td>ALCG</td>
<td>Add doubleword from memory</td>
<td>SLBG</td>
<td>Subtract doubleword from memory</td>
</tr>
<tr>
<td>ALCGR</td>
<td>Add doubleword from c(GG R2)</td>
<td>SLBGR</td>
<td>Subtract doubleword from c(GG R2)</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

    LM 0,1,A Load A in register pair
    AL 1,B+4 Logically add low-order part of B
    ALC 0,B Add high-order part of B with carry
    STM 0,1,Sum Store the double-length sum
    LM 0,1,A Get first operand
    SL 1,B+4 Logically subtract low-order parts
    SLB 0,B Subtract high-order parts with borrow
    STM 0,1,Diff Store 64-bit difference

- These instructions are especially useful for multi-precision arithmetic
Operations With Mixed 64-Bit and 32-Bit Operands

• 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 R₁

<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
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 R₁) to word from memory</td>
</tr>
<tr>
<td>CLR</td>
<td>Logical compare c(GR R₁) to word from c(GR R₂)</td>
</tr>
<tr>
<td>CLG</td>
<td>Logical compare c(GG R₁) to doubleword from memory</td>
</tr>
<tr>
<td>CLGR</td>
<td>Logical compare c(GG R₁) to doubleword from c(GG R₂)</td>
</tr>
<tr>
<td>CLGF</td>
<td>Logical compare c(GG R₁) to zero-extended word from memory</td>
</tr>
<tr>
<td>CLGFR</td>
<td>Logical compare c(GG R₂) to zero-extended word from c(GR R₂)</td>
</tr>
<tr>
<td>CLM, CLMY</td>
<td>Logical compare bytes from <em>low</em> half of c(GG R₁) to bytes in memory</td>
</tr>
<tr>
<td>CLMH</td>
<td>Logical compare bytes from <em>high</em> half of c(GG R₁) to bytes in memory</td>
</tr>
</tbody>
</table>

- The CC settings are the same as for other logical comparisons
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 \( R_1 \)

```
//CCFDUS///////
```

“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>8</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>

- SR  \( 0,0 \)  Set GRO to zero
- SPM  \( 0 \)  Set CC and Program Mask bits to zero
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
• Assume an \( n \)-bit register looks like this:

\[
\begin{array}{ccccccc}
\text{Bits Before Shifting} \\
\hline
\text{a} & \text{b} & \text{c} & \text{d} & \text{w} & \text{x} & \text{y} & \text{z} \\
0 & 1 & 2 & 3 & n-4 & n-3 & n-2 & n-1 \\
\end{array}
\]

• A \textit{unit shift} moves bits left or right by one position

• Logical shifts:

  - The Condition Code is unchanged
- Arithmetic shifts (“s” = sign bit):

- The Condition Code is set

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

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

  L  7,=X′87654321′  Initial contents of GR7
  SLL 7,5(0)  Results: $c(GR7) = X′ECA86420′$

  L  6,=X′87654321′  Initial contents of GR6
  SRL 6,5(0)  Results: $c(GR6) = X′043B2A19′$

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

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

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

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

- Exercise: Verify the example results
Double-Length Logical Shifts

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

\[
\begin{array}{cccccc}
\text{GR } R_1 & & & & \text{GR } R_1+1 \\
\begin{array}{cccccccc}
a & b & c & \ldots & l & m& \leftrightarrow & n & o & p & \ldots & y & z \\
\end{array}
\end{array}
\]

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

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

\[
\begin{array}{l}
\text{SR} 7,7 & \text{Set c(Gr7) to zero} \\
\text{SRDL} 6,5 & \text{Move rightmost 5 bits into GR7} \\
\text{LTR} 7,7 & \text{Are those bits zero?} \\
\text{BZ} \text{Yes_It_Is} & \text{If the bits are zero, it's a multiple} \\
\text{B} \text{Sorry_No} & \text{Sorry, it's not a multiple of 32}
\end{array}
\]
Arithmetic Shift Instructions

- Arithmetic shifts always set the CC:

<table>
<thead>
<tr>
<th>Operation</th>
<th>CC Setting and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left shift</td>
<td>0: Result is zero</td>
</tr>
<tr>
<td></td>
<td>1: Result is &lt; zero</td>
</tr>
<tr>
<td></td>
<td>2: Result is &gt; zero</td>
</tr>
<tr>
<td></td>
<td>3: Result has overflowed</td>
</tr>
<tr>
<td>Right shift</td>
<td>0: Result is zero</td>
</tr>
<tr>
<td></td>
<td>1: Result is &lt; zero</td>
</tr>
<tr>
<td></td>
<td>2: Result is &gt; zero</td>
</tr>
<tr>
<td></td>
<td>3: Cannot occur</td>
</tr>
</tbody>
</table>

- Examples:

  \[ L \ 6,=X'87654321' \]
  \[ SRA \ 6,5 \ c(\text{GR6}) = X'\text{FC3B2A19}', \ CC=1 \]
  \[ L \ 7,=X'87654321' \]
  \[ SLA \ 7,5 \ c(\text{GR7}) = X'ECA86420', \ CC=3 \text{ (overflow)} \]

- Remember: for left shifts, if a lost bit differs from the sign bit (s \neq b), an overflow exception is indicated
A Rotating shift looks like this (compare slide 32)

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

\[
\begin{align*}
L & \ 0,=A(X'56789ABC') & \text{Load initial data into GR0} \\
RLL & \ 1,0,10 & \text{Rotate 10 bits, result in GR1} \\
\end{align*}
\]

- Then \( c(GR1) = X'E26AF159' \)

\[
\begin{align*}
LG & \ 0,=AD(X'56789ABCDEF01234') & \text{Initialize GGO} \\
RLLG & \ 1,0,10 & \text{Rotate 10 bits, result in GG1} \\
\end{align*}
\]

- Then \( c(GG1) = X'E26AF37BC048D159' \)

- 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 B\textsubscript{2} register

\begin{verbatim}
L 9,ShiftAmt    Shift amount calculated previously
L 0,Data       Get data to be shifted
SLL 0,0(9)     Shift left by calculated amount
\end{verbatim}

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

• Example: calculate \(2^N\), where \(0 \leq N < 31\)

\begin{verbatim}
L 1,N           Get small integer N
L 0,=F'1'       Put a 1-bit at right end of GRO (2^0)
SLL 0,0(1)      Leave \(2^N\) in GRO
\end{verbatim}

- Exercise: What will happen if \(N \geq 31\)?
An extended form of length modifier lets you specify lengths in bits: put a period (.) after the modifier L

\[
\text{DC FL3'8' and DC FL.24'8' are equivalent (byte length) (bit length)}
\]

- A nominal value can be any length (subject to normal truncation and padding rules)

\[
\text{DC FL.12'2047', FL.8'64', XL.4'D' generates X'7FF40D'}
\]

- Incomplete bytes are padded with zero bits:

\[
\text{DC FL.12'2047' generates X'7FF0'}
\]

- Bit-length constants are useful for tightly packed data
Binary Multiplication and Division

- Multiplication notation:
  
  \[
  \begin{array}{c|c}
  \text{Multiplicand} & \text{First operand} \\
  \times & \text{Multiplier} \\
  \text{Product} & \text{Second operand} \\
  \text{Product} & \text{Single or double-length result}
  \end{array}
  \]

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

- Multiplying single-length operands usually requires a double-length register pair

- Division notation:
  
  \[
  \begin{array}{c|c}
  \text{Quotient} & \text{Dividend} \\
  \text{Divisor} & \text{Dividend} = \text{first operand} \\
  \text{Divisor} & \text{Divisor} = \text{second operand} \\
  \text{Remainder} & \\
  \end{array}
  \]

- Division by a single-length divisor usually requires a double-length dividend, producing single-length quotient and remainder

- Multiply and divide instructions do not change the CC
For 64-bit signed products of 32-bit operands:

<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>

Example(1): Square the number in GR1
MR 0,1 c(GR0,GR1) = c(GR1) * c(GR1)

Example(2): Square the number in GR0
LR 1,0 Copy c(GR0) to GR1 (odd register of pair)
MR 0,0 Multiply c(GR0) by c(GR1)
When multiplying small values, you can use a single register:

<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←32×16)</td>
<td>MSR</td>
<td>Multiply Single Register (32←32×32)</td>
</tr>
<tr>
<td>MS, MSY</td>
<td>Multiply Single (32←32×32)</td>
<td>MSG</td>
<td>Multiply Single Register (64←64×64)</td>
</tr>
<tr>
<td>MSG</td>
<td>Multiply Single (64←64×64)</td>
<td>MSGR</td>
<td>Multiply Single Register (64←64×64)</td>
</tr>
<tr>
<td>MSGF</td>
<td>Multiply Single (64←64×32)</td>
<td>MSGFR</td>
<td>Multiply Single Register (64←64×32)</td>
</tr>
</tbody>
</table>

Examples:

```
MH  5,=H'100'   c(GR5)*100
LH  1,N        Load c(N) into GR1
MSR 1,1        c(GR1)*c(GR1) = N^2
LG  1,=FD'12345678' c(GG1) = 12345678
MSG 1,=FD'23456789' c(GG1) = 289589963907942
LG  1,=FD'12345678' c(GG1) = 12345678
L  5,=F'23456789' c(GR5) = 23456789 (32 bits!)
MSGFR 1,5      c(GG1) = 289589963907942
```
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 × 32)</td>
<td>MLR</td>
<td>Multiply Logical Register (32 + 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 × 64)</td>
<td>MLGR</td>
<td>Multiply Logical Register (64 + 64 ← 64 × 64)</td>
</tr>
</tbody>
</table>

- Examples:

* Logical multiplication: \((2^{*32}) \times (2^{*32}) = 18446744065119617025\)

\[
\begin{align*}
L & 1, = F' - 1' & c(GR1) = X' FFFFFFFF'
\end{align*}
\]

\[
\begin{align*}
MLR & 0, 1 & c(GRO, GR1) = X' FFFFFFFE 00000001'
\end{align*}
\]

\[
\begin{align*}
LG & 1, = FD'74296604373' & c(GG1) = 74296604373
\end{align*}
\]

\[
\begin{align*}
MLG & 0, = FD'9876543210' & c(GG0, GG1) = 733793623446209457330
\end{align*}
\]
Division Instructions

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

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

- 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:

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_1+1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remainder</td>
<td>Quotient</td>
</tr>
</tbody>
</table>
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÷32)</td>
<td>DR</td>
<td>Divide Register (32,32←32+32÷32)</td>
</tr>
</tbody>
</table>

- The results are in an even-odd register pair

- Example

```
L 2,=F'–14352'  Dividend in GR2
SRDA 2,32       Create double-length dividend
D 2,=F'17'      c(GR2) = –4. c(GR3) = –844
```
The Divide Single instructions use a single-length dividend in an odd register; the results are as above.

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSG</td>
<td>Divide Single (64,64←64÷64)</td>
<td>DSGR</td>
<td>Divide Single Register (64,64←64÷64)</td>
</tr>
<tr>
<td>DSGF</td>
<td>Divide Single (64,64←64÷32)</td>
<td>DSGFR</td>
<td>Divide Single Register (64,64←64÷32)</td>
</tr>
</tbody>
</table>

Examples:

- \( \text{LG 5,=FD'12345678901'} \)  \( c(\text{GG1}) = 12345678901 \)
- \( \text{DSG 4,=FD'777'} \)  \( \text{Divide by 777 (64-bit divisor)} \)
- \( c(\text{GG4}) = 493 \) (remainder),  \( c(\text{GG5}) = 15888904 \) (quotient)
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÷32)</td>
<td>DLR</td>
<td>Divide Logical Register (32,32←32+32÷32)</td>
</tr>
<tr>
<td>DLG</td>
<td>Divide Logical (64,64←64+64÷64)</td>
<td>DLGR</td>
<td>Divide Logical Register (64,64←64+64÷64)</td>
</tr>
</tbody>
</table>

The dividend is always double-length.

Example:

```
L 0,=F′-2′       Set GRO to X′FFFFFFFFFE′
SR 1,1           Set GRI to X′00000000′
DL 0,=F′-1′      Divide logically by X′FFFFFFFF′
               * Quotient and remainder = X′FFFFFFFFFE′
```

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><strong>Operand 1 length</strong></td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td><strong>Operand 2 length</strong></td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic ×</strong></td>
<td>MH</td>
<td>MS</td>
<td>M</td>
<td>MSGF</td>
<td>MSG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MR</td>
<td>MSGFR</td>
<td>MSGR</td>
</tr>
<tr>
<td><strong>Logical ×</strong></td>
<td>ML</td>
<td>ML</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MLR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dividend length (bits)</strong></td>
<td>32 + 32</td>
<td>64</td>
<td>64 + 64</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Divisor length</strong></td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td><strong>Quotient &amp; remainder length</strong></td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic ÷</strong></td>
<td>D</td>
<td>DR</td>
<td>DSG</td>
<td>DSGR</td>
<td></td>
</tr>
<tr>
<td><strong>Logical ÷</strong></td>
<td>DL</td>
<td>DLR</td>
<td>DSGF</td>
<td>DSGFR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>NR</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>OG</td>
<td>OR (64)</td>
<td>OGR</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</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>
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’EDAB6567’</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></th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011</td>
<td>3 0011</td>
<td>3 0011</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9 1001</td>
<td>9 1001</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>B 1011</td>
<td>A 1010</td>
</tr>
</tbody>
</table>
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:

   \[ Y = X \text{ AND } (X - 1) \]

   - Example:

     \[
     \begin{align*}
     &L 0,=F'6' \quad X \text{ in GR0} \quad X'00000006' \\
     &LR 1,0 \quad \text{Copy } X \text{ to GR1} \quad X'00000006' \\
     &S 1,=F'1' \quad (X-1) \quad X'00000005' \\
     &NR 1,0 \quad (X-1) \text{ AND } X \quad X'00000004'
     \end{align*}
     \]

3. Isolate the rightmost 1-bit of a word

   \[ Y = X \text{ AND } (-X) \]

   - Example:

     \[
     \begin{align*}
     &L 0,=F'12' \quad X \text{ in GR0} \quad X'0000000C' \\
     &LCR 1,0 \quad \text{Copy } -X \text{ to GR1} \quad X'FFFFFFF4' \\
     &NR 1,0 \quad X \text{ AND } (-X) \quad X'00000004'
     \end{align*}
     \]
Chapter VI: Addressing, Immediate Operands, and Loops

This chapter describes three useful and important topics:

- Section 20 discusses ways the CPU can generate Effective Addresses, and how those addresses depend on the current addressing mode.

- Section 21 introduces instructions with *immediate* operands, where one of the operands of the instruction is contained in the instruction itself.

- Section 22 reviews instructions that help you manage loops: iterative execution of a block of instructions that perform some repeated action.
System z supports three types of address generation:

1. base-displacement with unsigned 12-bit displacements
   - This was described in Section 5
2. base-displacement with signed 20-bit displacements
3. relative-immediate.

... and three addressing \textit{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.
Address Generation: Signed Displacements

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

```
| opcode | R₁ | X₂ | B₂ | DL₂ | DH₂ | opcode |
```

- Address generation first creates a signed displacement:

```
Instruction ──┬───┬───┬───────────┬────────┬─ ─
─ ─┴───┴───┴─────┬─────┴───┬────┴─ ─
/SM630000/SV040000

Effective Address
```

- Addressing range is ±512K (vs. 4K for unsigned 12-bit displacements)
• Relative-immediate instructions have two basic formats:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>R1</th>
<th>op</th>
<th>RI2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>R1</td>
<td>op</td>
<td>RI2</td>
</tr>
</tbody>
</table>

• Address generation involves *signed* offsets:

```
<table>
<thead>
<tr>
<th>Opcode, regs</th>
<th>sbbbbbbbbbbbbbbb</th>
<th>RI-type instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sbbbbbbbbbbbbbbb</td>
<td>Shift left 1 bit</td>
</tr>
<tr>
<td></td>
<td>sbbbbbbbbbbbbbbb</td>
<td>64-bit signed offset</td>
</tr>
<tr>
<td></td>
<td>sbbbbbbbbbbbbbbb</td>
<td>(Not the PSW's IA!)</td>
</tr>
</tbody>
</table>
```

Add to

```
address of the instruction itself
```

Effective Address

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

• “Addressing Mode” is often abbreviated “AMode”

• Determines which bits of an Effective Address are used for addressing:

```
0  39  40  63
  ┌────────────────────────────────────────────────────┬─────────────────────────────────┐
  │ /SM630000── xxxxx .... xxxxx ... /SM630000────────────── ignored ───────────────────────────/SM590000 /SM630000─────── 24─bit address ───────/SM590000

0  33  63
  ┌───────────────────────────────────────────┬──────────────────────────────────────────┐
  │ /SM630000── xxxxx ..... xxxxx ... /SM630000────────────── ignored ──────────────────/SM590000 /SM630000──────────── 31─bit address ────────────/SM590000

0              63
  ┌──────────────────────────────────────────────────────────────────────────────────────┐
  │ AMode 64
  └──────────────────────────────────────────────────────────────────────────────────────┘ /SM630000───────────────────────────────────64─bit address ──────────────────────────────────/SM590000
```

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:

  $\begin{align*}
  \text{LA} & \ 0,n(0,0) & 0 \leq n \leq 4095 \\
  \text{LAY} & \ 0,n(0,0) & -2^{19} \leq n \leq 2^{19}-1 \\
  \end{align*}$

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

  $\begin{align*}
  \text{LAY} & \ 0,-1 & \text{Put } -1 \text{ in register } 0 \\
  \end{align*}$

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

- LARL always creates a memory address (relative to LARL)
This table summarizes the three Load Address instructions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Instruction</th>
<th>Result in R₁ general register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Address (based)</td>
<td>LA</td>
<td>AMode = 24: Effective Address in bits 40-63; zero in bits 32-39; bits 0-31 unchanged.</td>
</tr>
<tr>
<td></td>
<td>LAY</td>
<td>AMode = 31: Effective Address in bits 33-63; zero in bit 32; bits 0-31 unchanged.</td>
</tr>
<tr>
<td>Load Address (relative)</td>
<td>LARL</td>
<td>AMode = 64: Effective Address in bits 0-63.</td>
</tr>
</tbody>
</table>
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>RI</th>
<th>opcode</th>
<th>R1</th>
<th>op</th>
<th>I2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>RIL</th>
<th>opcode</th>
<th>R1</th>
<th>op</th>
<th>I2</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>HH</th>
<th>HL</th>
<th>LH</th>
<th>LL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>47</td>
<td>48</td>
<td>63</td>
</tr>
</tbody>
</table>

- **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>IIHF</td>
<td>Insert Logical Immediate (high) (64←32)</td>
<td>IILF</td>
<td>Insert Logical Immediate (low) (64←32)</td>
</tr>
<tr>
<td>IIHH</td>
<td>Insert Logical Immediate (high high) (64←16)</td>
<td>IIHL</td>
<td>Insert Logical Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>IILH</td>
<td>Insert Logical Immediate (low high) (64←16)</td>
<td>IILL</td>
<td>Insert Logical Immediate (low low) (64←16)</td>
</tr>
</tbody>
</table>

© IBM 2015 System z Assembler Language
• 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>LLIHF</td>
<td>Load Logical Immediate (high) (64←32)</td>
<td>LLILF</td>
<td>Load Logical Immediate (low) (64←32)</td>
</tr>
<tr>
<td>LLIHH</td>
<td>Load Logical Immediate (high high) (64←16)</td>
<td>LLIHL</td>
<td>Load Logical Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>LLILH</td>
<td>Load Logical Immediate (low high) (64←16)</td>
<td>LLILL</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)
Arithmetic Instructions with Immediate Operands

- Arithmetic and logical add and subtract instructions:

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHI</td>
<td>Add Halfword Immediate (32←16)</td>
<td>AGHI</td>
<td>Add Halfword Immediate (64←16)</td>
</tr>
<tr>
<td>AFI</td>
<td>Add Immediate (32)</td>
<td>AGFI</td>
<td>Add Immediate (64←32)</td>
</tr>
<tr>
<td>ALFI</td>
<td>Add Logical Immediate (32)</td>
<td>ALGFI</td>
<td>Add Logical Immediate (64←32)</td>
</tr>
<tr>
<td>SLFI</td>
<td>Subtract Logical Immediate (32)</td>
<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>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHI</td>
<td>Compare Halfword Immediate (32←16)</td>
<td>CGHI</td>
<td>Compare Halfword Immediate (64←16)</td>
</tr>
<tr>
<td>CFI</td>
<td>Compare Immediate (32)</td>
<td>CGFI</td>
<td>Compare Immediate (64←32)</td>
</tr>
<tr>
<td>CLFI</td>
<td>Compare Logical Immediate (32)</td>
<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>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHI</td>
<td>Multiply Halfword Immediate (32←16)</td>
<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>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIHF</td>
<td>AND Immediate (high) (64←32)</td>
<td>NILF</td>
<td>AND Immediate (low) (64←32)</td>
</tr>
<tr>
<td>NIHH</td>
<td>AND Immediate (high high) (64←16)</td>
<td>NIHL</td>
<td>AND Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>NILH</td>
<td>AND Immediate (low high) (64←16)</td>
<td>NILL</td>
<td>AND Immediate (low low) (64←16)</td>
</tr>
<tr>
<td>OIHF</td>
<td>OR Immediate (high) (64←32)</td>
<td>OILF</td>
<td>OR Immediate (low) (64←32)</td>
</tr>
<tr>
<td>OIHH</td>
<td>OR Immediate (high high) (64←16)</td>
<td>OIHL</td>
<td>OR Immediate (high low) (64←16)</td>
</tr>
<tr>
<td>OILH</td>
<td>OR Immediate (low high) (64←16)</td>
<td>OILL</td>
<td>OR Immediate (low low) (64←16)</td>
</tr>
<tr>
<td>XIHF</td>
<td>XOR Immediate (high) (64←32)</td>
<td>XILF</td>
<td>XOR Immediate (low) (64←32)</td>
</tr>
</tbody>
</table>
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></td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td><strong>Operand 2</strong></td>
<td>AHI</td>
<td>AFI</td>
<td>AGHI</td>
</tr>
<tr>
<td>Arithmetic Add/Subtract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Add/Subtract</td>
<td></td>
<td>ALFI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SLFI</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Compare</td>
<td></td>
<td>CFI</td>
<td></td>
</tr>
<tr>
<td>Logical Compare</td>
<td></td>
<td>CLFI</td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>MHI</td>
<td></td>
<td>MGHI</td>
</tr>
</tbody>
</table>
This section describes three powerful types of branch instruction:

- **Branch Relative on Condition**: these are similar to the familiar Branch on Condition instructions, but with a relative-immediate operand address
- **Branch on Count**: these help control the execution of loops controlled by the number of iterations
- **Branch on Index**: these powerful instructions can increment an index value, compare the sum to an end value, and determine whether or not to branch, all in a single instruction

We will also examine some general styles of loop organization
The BRC and BRCL instructions have these formats:

\[
\begin{array}{cccc}
A7 & M1 & 4 & RI2 \\
\end{array}
\]

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

\[
\begin{array}{cccc}
C0 & M1 & 4 & RI2 \\
\end{array}
\]

- 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
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>JLC</td>
<td>$M_1$ Conditional Branch</td>
</tr>
<tr>
<td>BRU</td>
<td>J</td>
<td>JLU</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BRNO</td>
<td>JNO</td>
<td>JLNO</td>
<td>Branch if No Overflow</td>
</tr>
<tr>
<td>BRNH</td>
<td>JNH</td>
<td>JLNH</td>
<td>Branch if Not High</td>
</tr>
<tr>
<td>BRNP</td>
<td>JNP</td>
<td>JLP</td>
<td>Branch if Not Plus</td>
</tr>
<tr>
<td>BRNL</td>
<td>JNL</td>
<td>JLNL</td>
<td>Branch if Not Low</td>
</tr>
<tr>
<td>BRNM</td>
<td>JNM</td>
<td>JLNM</td>
<td>Branch if Not Minus</td>
</tr>
<tr>
<td>BRE</td>
<td>JE</td>
<td>JLE</td>
<td>Branch if Equal</td>
</tr>
<tr>
<td>::</td>
<td>::</td>
<td>::</td>
<td></td>
</tr>
<tr>
<td>BRP</td>
<td>JP</td>
<td>JLP</td>
<td>Branch if Plus</td>
</tr>
<tr>
<td>BRO</td>
<td>JO</td>
<td>JLO</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:

```
LHI 2,10              Count of numbers in the table
XR 1,1               Set index to zero
XR 0,0               Set sum to zero
Add A 0,Table(1)      Add an integer from the table
AHI 1,4              Increment index by number length
AHI 2,—1             Reduce count by 1
JNZ Add               If not zero, repeat
ST 0,Sum             Store the resulting sum

Table DC F’1,2,3,4,5,6,7,8,9,10’
```

• We used c(GR1) as the “index” to reference each item in turn
Count-controlled loops are easily managed with these instructions

<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,</td>
<td>Branch Relative on Count (32)</td>
<td>BRCTG,</td>
<td>Branch Relative on Count (64)</td>
</tr>
<tr>
<td>JCT</td>
<td></td>
<td>JCTG</td>
<td></td>
</tr>
</tbody>
</table>

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

Example: add the numbers from 1 to 10 (in reverse order)

| XR    | 0,0                                | Clear GRO for the sum |
| LA    | 1,10                               | Number of values to add |
| Repeat| AR 0,1                             | Add a value to the sum |
| Repeat| BCT 1,Repeat                       | Reduce counter by 1, repeat if nonzero |
| ST    | 0,Sum                              | Store the result for display |
There are many types of loop; some of the most common are:

1. “Do-Until”

   - Initialize index, increment, comparand
   - Loop body
   - Add increment to index
   - Compare to comparand
   - Done
   - Not done

2. “Do-While”

   - Initialize index, increment, comparand
   - Compare to comparand
   - Not done
   - Loop body
   - Add increment to index
   - Done

3. A combination

   - Initialize
   - Loop body, first part
   - Exit test
   - Loop body, remainder
   - Done
Branch on Index Instructions

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

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BXH</td>
<td>Branch on Index High (32)</td>
<td>BXHG</td>
<td>Branch on Index High (64)</td>
</tr>
<tr>
<td>BXLE</td>
<td>Branch on Low or Equal (32)</td>
<td>BXLEG</td>
<td>Branch on Index Low or Equal (64)</td>
</tr>
<tr>
<td>BRXH, JXH</td>
<td>Branch Relative on Index High (32)</td>
<td>BRXHG, JXHG</td>
<td>Branch Relative on Index High (64)</td>
</tr>
<tr>
<td>BRXLE, JXLE</td>
<td>Branch Relative on Low or Equal (32)</td>
<td>BRXLG, JXLEG</td>
<td>Branch Relative on Index Low or Equal (64)</td>
</tr>
</tbody>
</table>

© IBM 2015 System z Assembler Language
• For all the “Branch on Index” instructions:

• The $R_1$ operand is the index; the increment is in $R_3$, and the comparand is in $R_3|1$ ($R_3$ value with low-order bit forced to 1)

• Examples

  1. Add the numbers in a table with a two-instruction loop

     XR 0,0          Set sum to zero
     XR 1,1          Set index to zero
     LM 2,3,=F′4,36′  Initialize increment, comparand
     Add A 0,Table(1) Add an integer from the table
     BXLE 1,2,Add    Increment index, compare to 36

     Table DC F′1,2,3,4,5,6,7,8,9,10′

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

     XR 0,0          Clear sum to zero
     LHI 1,1         Initialize “index” to 1
     LM 2,3,=F′1,10′ Initialize increment and comparand
     Add AR 0,1      Add “index” to sum
     BXLE 1,2,Add    Repeat 10 times
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(1) Add an element of the table
     BXH 1,3,Add Decrease index, compare to −4
     Table DC F’3,1,4,1,5,9,2,6,5,89’
  2. Calculate a table of cubes of the first 10 integers
     LA 7,10 Initial value of N is 10
     LA 4,36 Initial index = 36
     LHI 5,−4 Increment and comparand are −4
     Mult LR 1,7 N
     MR 0,7 N squared
     MR 0,7 N cubed
     ST 1,Cube(4) Store in table
     BCTR 7,0 Decrease N by 1
     BXH 4,5,Mult Count down and loop
Specialized Uses of BXLE and BXH

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

  1. Branch to XXX if c(GR4) is $\leq 0$

     \[
     \begin{align*}
     &XR \quad 9,9 \quad \text{Set GR9 to zero} \\
     &BXLE \quad 4,9,XXX \quad \text{Branch to XXX if c(GR4) is } \leq 0 \\
     \text{and...}! \\
     &XR \quad 9,9 \quad \text{Set GR9 to zero} \\
     &BXH \quad 4,9,YYY \quad \text{Branch to YYY if c(GR4) is } > 0
     \end{align*}
     \]

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

     \[
     \begin{align*}
     &LHI \quad 7,1 \quad \text{Initialize GR7 to +1} \\
     &BXH \quad 2,7,XXX \quad \text{Increment c(GR2), branch to XXX}
     \end{align*}
     \]

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

     \[
     \text{BXH} \quad 5,4,ZZZ
     \]
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>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></td>
<td>BXLE</td>
</tr>
<tr>
<td>Branch on Index (Relative)</td>
<td>BRXH</td>
</tr>
<tr>
<td></td>
<td>BRXLE</td>
</tr>
</tbody>
</table>
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 $\leftarrow$ $I_2$</td>
<td>No</td>
</tr>
<tr>
<td>AND</td>
<td>NI, NIY</td>
<td>Operand 1 $\leftarrow$ Operand 1 AND $I_2$</td>
<td>Yes</td>
</tr>
<tr>
<td>OR</td>
<td>OI, OIY</td>
<td>Operand 1 $\leftarrow$ Operand 1 OR $I_2$</td>
<td>Yes</td>
</tr>
<tr>
<td>XOR</td>
<td>XI, XIY</td>
<td>Operand 1 $\leftarrow$ Operand 1 XOR $I_2$</td>
<td>Yes</td>
</tr>
<tr>
<td>Compare</td>
<td>CLI, CLIY</td>
<td>Operand 1 Compared to $I_2$</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>$I_2$</th>
<th>$B_1$</th>
<th>$D_1$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>$I_2$</th>
<th>$B_1$</th>
<th>$DL$</th>
<th>$DH$</th>
<th>opcode</th>
</tr>
</thead>
</table>

- Write the operand field as either $D_1(B_1),I_2$ or $S_1,I_2$
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 \( I_1 \) 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  
   NI X,B'11111101'  
   Same as 'MVI X,0' except CC set to 0  
   Sets bit 6 at X to 0

(b) OI X,255  
   OI X,B'00000010'  
   Same as 'MVI X,255' except CC set to 1  
   Sets bit 6 at X to 1

(c) OI LowerA,C'  
    LowerA DC C'a'  
    c(LowerA) now is C'A'  
    Initially, lower case letter 'a'

(c) XI X,B'00000010'  
    Inverts bit 6 at X
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 = I₂</td>
</tr>
<tr>
<td>1</td>
<td>Operand 1 &lt; I₂</td>
</tr>
<tr>
<td>2</td>
<td>Operand 1 &gt; I₂</td>
</tr>
</tbody>
</table>

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

CLI =C’ A’, X’ C1’  CC = 0:  c(Operand 1) = I₂
CLI =X’00’,0        CC = 0:  c(Operand 1) = I₂
CLI =C’ , B’01000000’  CC = 0:  c(Operand 1) = I₂
CLI =X’1’,X’2’       CC = 1:  c(Operand 1) < I₂
CLI =C’ A’,250       CC = 1:  c(Operand 1) < I₂
CLI =C’ X’, C’ X’ –1 CC = 2:  c(Operand 1) > I₂
CLI =X’1’,X’0’       CC = 2:  c(Operand 1) > I₂
Test Under Mask Instructions

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Instruction</th>
<th>Mnem</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td>Test Under Mask</td>
<td>TMY</td>
<td>Test Under Mask</td>
</tr>
</tbody>
</table>

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

<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>

**Mnem**  
**Instruction**

<table>
<thead>
<tr>
<th>TM</th>
<th>Num,X'80'</th>
<th>Test leftmost bit of a number</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Minus</td>
<td>Branch if a 1-bit, it's negative</td>
</tr>
<tr>
<td>TM</td>
<td>Num+L Num-1,1</td>
<td>Test rightmost bit of a number</td>
</tr>
<tr>
<td>JZ</td>
<td>Even</td>
<td>Branch if low-order bit is zero</td>
</tr>
<tr>
<td>TM</td>
<td>BB,255</td>
<td>Test all eight bits</td>
</tr>
<tr>
<td>JM</td>
<td>Mixed</td>
<td>Branch if not all zeros or all ones</td>
</tr>
</tbody>
</table>
- It’s better to *name* bit-data items than to use bit numbers or bit masks
- Discouraged techniques:
  - **O1** Person_X,128 Person has retired [Poor method]
  - or
  - **O1** Person_X,Bit1 Person has retired [Poor method]
- Better technique: name each flag bit separately
  - Retired Equ X’40’ Retired status flag bit
  - FullTime Equ X’20’ Full time worker status flag bit
  - PartTime Equ X’10’ Part time worker status flag bit
  - Exempt Equ X’08’ Exempt employee status flag bit
  - Hourly Equ X’04’ Hourly employee status flag bit
  - etc.
  - **O1** Person_X,Retired Person has retired [Better Method]
Avoiding Bit-Naming Problems

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

  Flag1 DS X │ Flag2 DS X  
  Bit0 Equ X’80’ │ Bit1 Equ X’40’

- Normally we would write something like

  OI Flag1,Bit0 │ OI Flag2,Bit1

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

  OI Flag2,Bit0 │ OI Flag1,Bit1

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

  Fulltime Equ *,X’20’  
  PartTime Equ *,X’10’  
  ─ ─ ─DS X  

  Now define the (unnamed) owning byte

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

  TM Fulltime,L’Fulltime [Best!]
Instruction Modification

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

  1. Skip some statements after they’re executed once

          NOP   NOP   SkipIt     Fall through first time here
          OI   NOP+1,X’F0’       Change NOP to unconditional branch

          SkipIt   DC   OH      Continue execution here

  2. Alternating between branching or not

          XI   Switch+1,X’F0’    Alternate branch masks at ‘Switch’
          Switch   BC   15,SomeWhereElse Mask = 0, 15, 0, 15, ...

• This is a poor practice:

  1. Serious negative impact on performance

  2. The program can’t be shared in memory

  3. You may not be debugging the program in the listing

• Advice: use a bit flag in a data area
• These are the Storage-Immediate instructions described in Section 23:

<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>
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>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>
<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:

  \[ \text{mnemonic} \ D₁(N,B₁),D₂(B₂) \]

- We’ll see how \( L \) and \( N \) differ in Section 24.5
- Only TRT and TRTR change general registers (only GR1, GR2)
A machine instruction operand can have one of 3 formats:

\[ \text{expr} \quad \text{expr(expr)} \quad \text{expr(expr,expr)} \text{ or expr(,expr)} \]

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:

<table>
<thead>
<tr>
<th>Explicit Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1(N),S_2 )</td>
</tr>
<tr>
<td>( D_1(N,B_1),S_2 )</td>
</tr>
<tr>
<td>( S_1(N),D_2(B_2) )</td>
</tr>
<tr>
<td>( D_1(N,B_1),D_2(B_2) )</td>
</tr>
</tbody>
</table>

Examples:

- MVC \( \text{BB}(23), AA \) \( S_1(N),S_2 \)
- MVC \( \text{X'}47D'(23,9), AA \) \( D_1(N,B_1),S_2 \)
- MVC \( \text{BB}(23), \text{X'}125'(9) \) \( S_1(N),D_2(B_2) \)
- MVC \( 1149(23,9), 293(9) \) \( D_1(N,B_1),D_2(B_2) \)
Symbol Length Attribute References

- Written as \( L' \) followed by a symbol

\[
\begin{align*}
\text{LA} & \quad 0, L' \text{BB} & \quad C(\text{GR}) = \text{Length Attribute of BB}
\end{align*}
\]

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

\[
\begin{align*}
G & \quad \text{Equ} \quad * & \quad \text{Length attribute of G} = 1 \\
H & \quad \text{Equ} \quad *,20 & \quad \text{Length attribute of H} = 20
\end{align*}
\]

2. Literals have the length attribute of the first operand

\[
\begin{align*}
\text{LA} & \quad 0, =X'123456,ABC,FEDCBA98' & \quad c(\text{GR0}) = 3
\end{align*}
\]

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

\[
\begin{align*}
\text{MVC} & \quad \text{BB(L')} , AA & \quad \text{Length attribute of MVC} = 6
\end{align*}
\]
Implied Lengths

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

<table>
<thead>
<tr>
<th>Implied Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1, S_2$</td>
</tr>
<tr>
<td>$D_1(B_1), S_2$</td>
</tr>
<tr>
<td>$S_1, D_2(B_2)$</td>
</tr>
<tr>
<td>$D_1(B_1), D_2(B_2)$</td>
</tr>
</tbody>
</table>

- Implied lengths simplify specifying how many bytes are involved

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

- Summary of explicit/implied addresses and lengths

<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>
• 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
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 \((0 \leq L \leq 255)\)
- Examples of MVC

  MVC AA(23),BB Move 23 bytes from BB to AA
  MVC AA,BB Move L’AA bytes from BB to AA
  MVI Line,C’ ‘ Move a single blank to Line
  MVC Line+1(120),Line Propagate blanks to fill 121 bytes
  MVC Str(40),Str+2 Shift 40 bytes left 2 positions
  MVC Str+40(2),=C’ ‘ Replace last two bytes at Str by spaces

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

  MVCIN RevData,Data+L’Data─1 Move reversed from Data to Revdata

  Data DC C’12345’ Source operand
  RevData DC CL(L’Data) Target operand; result = C’54321’
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:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC W,W</td>
<td>AND each byte to itself</td>
</tr>
<tr>
<td>JZ Z</td>
<td>Branch if all bytes are zero</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OC W,W</td>
<td>OR each byte to itself</td>
</tr>
<tr>
<td>JZ Z</td>
<td>Branch if all bytes are zero</td>
</tr>
</tbody>
</table>

- **XOR**: set the at W to zero:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC W,W</td>
<td>XOR each byte with itself</td>
</tr>
</tbody>
</table>
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
  JE AllBlank Branch if equal

  or

  CLC =CL120',Line Compare to 120 blanks
  JE AllBlank Branch if equal

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

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

BlankTbl DC 240C',C'0123456789',6C' ' Translate table
Data DC C'A1G2p3?4+5W6/7' Result = C'1 2 3 4 5 6 7'
```
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
  NumTable DC 240X’0’,10X’1’,6X’0’
  Data DC C’A1G2p3?4+5W6/7’

  Detect numeric characters
  Data to be scanned for numerics
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
Example 1: Move a message to Line whose address and length are in GR8 and GR9 respectively

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

MoveMsg MVC Line(*--*),0(8) Move text at GR8 address to Line
```

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

```
L 1,Mask Get mask value
SLL 1,4 Position correctly for use as M1
EX 1,BCInst Execute the BC

NotMet Fall through if condition not met

BCInst BC 0,CondMet BC with mask of 0
```

Note that if the branch condition is met, control will be taken from the EX instruction
### Instructions discussed in Section 24:

<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>
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
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>

- Both instructions use Method “A” when interrupted, and two even-odd register pairs
  - The even-numbered register holds the operand address
  - The odd-numbered register holds the true operand length (0-2^{24} – 1 bytes)
  - The operands may have different lengths
- The high-order byte of \( R_2 + 1 \) holds a pad byte
- All four registers may be updated by the instructions
- Both instructions set the CC
  - MVCL sets CC=3 and moves no data if destructive overlap is possible:
    - part of the target field is used as source data after data has been moved into it
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 \( c(R_1+1) \) is 0 before the source length \( c(R_2+1) \), set CC=1
  4. If the source length \( c(R_2+1) \) is 0 before the target length \( c(R_1+1) \), use the pad character as source data until the target length is 0; set CC=2

<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>Destructive Overlap, no data movement</td>
</tr>
</tbody>
</table>

- Example: Set 2400 bytes at Field to zeros

```
LA 0,Field c(R_1) = Target address
LHI 1,2400 c(R_1+1) = Target length
SR 3,3 c(R_2+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
```
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 \( c(R_1) = \text{Target address} \)
LHI 1,2400 \( c(R_1+1) = \text{Target length} \)
SR 3,3 \( c(R_2+1) = \text{Source length} = 0; \text{pad} = X'00' \)
* No source address is required if source length is zero
CLCL 0,2 \( \text{Compare target Field bytes to X'00'} \)
JE Cleared \( \text{Branch if the Field was all zeros} \)
```
Move Long and Compare Logical Long Extended

Mnemonic | Instruction | Mnemonic | Instruction
---|---|---|---
MVCLE | Move Long Extended | CLCLE | Compare Logical Long Extended

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>opcode</th>
</tr>
</thead>
</table>

- The R₁ and R₃ operands are like R₁ and R₂ for MVCL and CLCL; addresses and lengths depend on addressing mode
  - The low-order byte of operand 2 is the pad character (not an address!)
  - The odd-numbered registers hold 32- or 64-bit lengths (depending on addressing mode) vs. 24-bit lengths for MVCL/CLCL

- Assembler Language syntax
  
  mnemonic R₁,R₃,D₂(B₂) Target,source,pad_character

  as in

  MVCLE 2,8,C′′(0) Pad character = C′′
  CLCLE 4,14,X′40′(0) Pad character = X′40′
• Conceptually, MVCLE works like this:
  1. As each byte is moved, increment addresses, decrement lengths
  2. If both lengths=0 at the same time, set CC=0
  3. If the target length $c(R_1 + 1)$ is 0 before the source length $c(R_3 + 1)$, set CC=1
  4. If the source length $c(R_3 + 1)$ is 0 before the target length $c(R_1 + 1)$, use the pad character as source data until the target length is 0; set CC=2

<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

    LA   0,Field               c(R_1) = Target address
    LHI  1,2400                c(R_1 + 1) = Target length
    SR   3,3                   c(R_3 + 1) = Source length = 0
    SR   5,5                   c(R5) = Pad byte = X’00’

* No source address is required if source length is zero

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

```
LA 0,Field c(R_1) = Target address
LHI 1,2400 c(R_1+1) = Target length
SR 3,3 c(R_2+1) = Source length = 0
SR 5,5 Set pad character to X'00'
JE Cleared Branch if the Field was all zeros

* No source address is required if source length is zero
```

© IBM 2015
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 C’A C-string.’, X’0’ Generates ‘A C-string.n’

- 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 R<sub>1</sub>, R<sub>2</sub>

- 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
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,C’(’ Test character
  LA 4,Expr Start of string
  LA 8,Expr+L’ Expr One byte past end of string
  SRST 8,4 Search for ’(’ at Expr
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 $R_1$
  3. The source-string address is placed in $R_2$

• 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; $R_1$ points to end of first operand</td>
</tr>
<tr>
<td>3</td>
<td>Incomplete move; $R_1$ and $R_2$ 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 $R_1$ operand, translate table address is the $R_2$ operand
     - The string length is in (odd register) $R_1 + 1$
  2. GR0 is zeroed; the test character is placed in its rightmost byte
  3. TRE stops when (a) all bytes are translated, or (b) a source byte (which is not translated) matches the stop character

• Condition Code settings:

<table>
<thead>
<tr>
<th>CC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All bytes translated; $R_1$ incremented by length, $R_1+1$ set to 0</td>
</tr>
<tr>
<td>1</td>
<td>$R_1$ points to the byte matching the stop character; $R_1+1$ decremented by the number of bytes processed before the match</td>
</tr>
<tr>
<td>3</td>
<td>$R_1$ incremented and $R_1+1$ decremented by the number of bytes processed</td>
</tr>
</tbody>
</table>
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
  
  'ABCDEFG' and 'QRSDEFT' – matching strings at offset 3: lengths 1, 2, or 3
  
  'ABC' and 'BCD**' – if pad='*', matching substring at offset 3: length 2
  
  1. Operand addresses are in even-numbered registers 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>
• 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.
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
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
• The worldwide use of IBM “mainframes” required added character support
  - Each set of 256 encodings is called a Code Page
  - “Assembler Language EBCDIC” is Code Page 037

• Other EBCDIC code pages support national characters like á, ä, ç, Ø, Ω
  - Many characters have different encodings in different code pages

• The “Syntactic Character Set” has the same encodings across EBCDIC code pages:
  - blank, decimal digits, lower and upper case alphabetics, and
  - + < = > % & * ’ ( ) , _ – . / : ; ?
  - It does not include # @ $ (allowed in Assembler Language symbols)

• All modern EBCDIC code pages support the “euro” character €
• Widely used on non-System z computers
  - Basic encoding is 7 bits wide (X’00’-X’7F’)
    - First 32 positions (X’00’-X’1F’) are reserved for control codes

• Decimal digits are X’30’-X’39’
• Upper-case letters X’41’-X’5A’; lower-case X’61’-X’7A’
• For ASCII character constants, use subtype A:

\[
\text{DC CA'ASCII'} \quad \text{Generates X'4153434949'}
\]
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 db SI sb sb
```

- Assembling DBCS data requires the DBCS option
  - G-type constants and self-defining terms may be needed

- Used most often for representing Japanese characters
All Unicode characters can have any of 3 formats:
- UTF-8: an encoding is 1-4 bytes long
- UTF-16: most characters are 2 bytes long; some are 2 2-byte pairs
- UTF-32: all characters are 4 bytes long

Instructions can convert any encoding to any other

UTF-16 is most widely used; notation is U+nnnn where “nnnn” is 4 hex digits
- The encoding of U+nnnn is X’nnnn’
- ASCII encodings have values from U+0000 to U+00FF
  - Encodings U+0000 - U+FFFF are known as the “Basic Multilingual Plane”

Unicode constants are written with type extension U, and generate UTF-16 characters:

\[ DC \text{ CU'Unicode } \text{ Generates X'0055006E00690063 006F006400650020'} \]
We will discuss the instructions in three groups:

1. String search, compare, and move 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>

2. Translation instructions

<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>

3. Format conversion instructions

<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>
• These instructions are equivalent to the similar single-byte instructions (SRST, MVCLE, CLCLE)
  But they handle pairs of bytes (which need not be Unicode characters, nor halfword aligned)

• **SRSTU**: scan a string addressed by $R_2$ for a byte pair matching the rightmost 2 bytes of GR0; $R_1$ has the address of the first byte after the string
  - $R_2$ incremented by 2 for each comparison

• **MVCLU**: moves pairs of bytes from area addressed by $R_3$ to area addressed by $R_1$; lengths in $R_1+1$, $R_3+1$; “padding pair” is low-order 16 bits of $D_2(B_2)$
  **Effective Address**
  - Each pair moved increments addresses by 2, decrements lengths by 2
  - Special padding rules if any length is odd

• **CLCLU**: registers and “padding pair” assigned like MVCLU’s
  - Each pair compared increments addresses by 2, decrements lengths by 2
  - Lengths must be even
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></td>
<td>/// = zeroed by HLASM</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>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>M3</td>
<td>///</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td>Omit M₃ operand to act like RRE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assembler instruction format:

  \[
  \text{mnemonic } R₁,R₂[,M₃] \quad [ ] \text{indicates optional operand}
  \]
• Sometimes need to translate to, from, or among Unicode encodings

• Four instructions:

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

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

  **TRTO:** Convert 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

  \[ \text{TRxx } R_1, R_2[,M_3] \]

• 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, CUUTUF</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:

  \[ \text{CUxx} \quad R_1, R_2[,M_3] \quad \text{For CU12, CU14, CU21, CU24} \]
  \[ \text{CUxx} \quad R_1, R_2 \quad \text{For CU41, CU42} \]

  - Initial implementations (CUTFU, CUUTUF) did no “well-formedness” tests
  - If \( M_3 = 1 \), invalid operand data sets \( CC = 2 \)
• Each instruction uses an optional operand to create six “additional” instructions with a single opcode

• Operand format:

  Mnemonic $R_1, R_2[, M_3]$ $M_3$ bits are B’AFLO’

• $M_3$ mask bits:

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

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

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

• Mask bits provide greater flexibility
  - But not all 9 A-F-L combinations are meaningful...
• Suppose a 32-bit integer $X'12345678'$ starts at address $X'2400'$.

\[
\begin{array}{cccc}
12 & 34 & 56 & 78 \\
\end{array}
\]

IBM System z ("Big-Endian")

\[
\begin{array}{cc}
2400 & 2403 \\
\end{array}
\]

• On some processors (e.g. Intel) it’s stored like this:

\[
\begin{array}{cccc}
78 & 56 & 34 & 12 \\
\end{array}
\]

Some workstations ("Little-Endian")

\[
\begin{array}{cc}
2400 & 2403 \\
\end{array}
\]

<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>LRVR</td>
<td>Load Register Reversed (32)</td>
</tr>
<tr>
<td>LRVG</td>
<td>Load Reversed (64)</td>
<td>LRVGR</td>
<td>Load Register Reversed (64)</td>
</tr>
<tr>
<td>LRVH</td>
<td>Load Halfword Reversed (16)</td>
<td>STRVH</td>
<td>Store Halfword Reversed (16)</td>
</tr>
<tr>
<td>STRV</td>
<td>Store Reversed (32)</td>
<td>STRVG</td>
<td>Store Reversed (64)</td>
</tr>
</tbody>
</table>
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.
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
  
  \[
  \begin{align*}
  \text{ZCon1 DC} & \quad Z'1470369258' \quad \text{Generates X'} F1F4F7F0F3F6F9F2F5C8' \\
  \text{ZCon2 DC} & \quad Z' -1' \quad \text{Generates X'} D1' \\
  \text{ZCon3 DC} & \quad Z' 5, +6, -749' \quad \text{Generates X'} C5C6F7F4D7' \\
  \text{ZCon4 DC} & \quad ZL5' 29' \quad \text{Generates X'} F0F0F0F2C9'
  \end{align*}
  \]

- Only a length modifier is valid (no integer, scale, exponent)

- Decimal points in nominal values are ignored
  
  \[
  \begin{align*}
  \text{ZCon5 DC} & \quad Z' 1234.5' \quad \text{Generates X'} F1F2F730F4C5' \\
  \text{ZCon6 DC} & \quad Z' 1.2345' \quad \text{Generates X'} F1F2F730F4C5'
  \end{align*}
  \]

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

\[
\begin{array}{cccccccccccc}
| d & d & d & d & d & d & d & d & d | \\
\end{array}
\]

\[\text{d}=\text{decimal digit, } S=\text{sign code}\]

- 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>39</td>
<td>X’039C’</td>
</tr>
</tbody>
</table>
Packed Decimal Constants

• Defined with constant type P

<table>
<thead>
<tr>
<th>Constant</th>
<th>DC</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon1</td>
<td>DC P'12345'</td>
<td></td>
<td>Generates X'12345C'</td>
</tr>
<tr>
<td>PCon2</td>
<td>DC P'-27,+62'</td>
<td></td>
<td>Generates X'047D062C'</td>
</tr>
<tr>
<td>PCon3</td>
<td>DC PL4'999'</td>
<td></td>
<td>Generates X'0000999C' (Padded on left)</td>
</tr>
<tr>
<td>PCon4</td>
<td>DC PL2'12345'</td>
<td></td>
<td>Generates 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>Constant</th>
<th>DC</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCon5</td>
<td>DC P'1234.5'</td>
<td></td>
<td>Generates X'12345C'</td>
</tr>
<tr>
<td>PCon6</td>
<td>DC P'1.2345'</td>
<td></td>
<td>Generates 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
Converting Between Packed and Zoned

- Use the **PACK** and **UNPK** instructions
- Both have Assembler Language syntax
  
  \[ \text{mnemonic } D_1(N_1,B_1), D_2(N_2,B_2) \]

- Machine instruction format:

<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>
</table>

- 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>S₁</td>
<td>S₁(N₁)</td>
</tr>
<tr>
<td>Explicit Address</td>
<td>D₁(B₁)</td>
<td>D₁(N₁,B₁)</td>
</tr>
</tbody>
</table>
The PACK Instruction

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

  PACK Target, Source

  ┌─────┬─────┬─────┬─────┬─────┐
  │ Z d │ Z d │ Z d │ Z d │ S d │
  └───┬─┴───┬─┴───┬─┴───┬─┴─┬─┘
  │ │ │ │ ┌─┘ │ │ │ │ ┌─────┘ │ │
  │ │ │ │ ┌───┘/SV040000/SV040000/SV040000/SV040000/SV040000
  ┌─────┬─────┬─────┐│ d d │ d d │ d S │ Packed target (first) operand
  └─────┴─────┴─────┘

- 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!
The UNPK Instruction

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

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

  \[
  \begin{array}{cccc}
  \text{d d} & \text{d d} & \text{d S} \\
  \end{array}
  \]

  Packed source (second) operand

  \[
  \begin{array}{cccc}
  \text{Z d} & \text{Z d} & \text{Z d} & \text{Z d} & \text{S d} \\
  \end{array}
  \]

  Zoned target (first) operand

- 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!
Packing and Unpacking ASCII and Unicode Data

Pack ASCII: PKA PDTTarget,ASCII_Source(N)
Pack Unicode: PKU PDTTarget,Unicode_Source(N)
Unpack ASCII: UNPKA ASCII_Target(N),PDSource
Unpack Unicode: UNPKU Unicode_Target(N),PDSource

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

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

- L is the Encoded Length of the character operand
  Must be odd for Unicode (even number of bytes)

- For packing (PKA, PKU): L is length-1 of second (character) operand
  For unpacking (UNPKA, UNPKU): L is length-1 of first (character) operand
It often helps to display data in hex

Use UNPK and TR in these steps (we’ll assume 4-byte data):

1. Move source data to right half of a work area:
   \[
   \text{MVC WorkArea+4(4),SourceData}
   \]
   \[
   \text{WorkArea DS CL8,X } \quad \text{← The extra byte is important!}
   \]

2. Unpack one extra byte (at the right end):
   \[
   \text{UNPK WorkArea(9),WorkArea+4(5) \quad \text{Extra byte is swapped}}
   \]

3. Translate the “spread hex” to EBCDIC characters
   \[
   \text{TR WorkArea,=C’0123456789ABCDEF’–C’0’}
   \]

4. 8 bytes at WorkArea are ready for display or print
• Usually gives expected results
  - Operand size limitations for some operations

• Notation corresponds to internal representations

  Packed:  
  
<table>
<thead>
<tr>
<th>12</th>
<th>34</th>
<th>56</th>
<th>7D</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>34</td>
<td>56</td>
<td>7D</td>
</tr>
</tbody>
</table>

  written 1234567-

  Zoned:  
  
<table>
<thead>
<tr>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>C4</td>
</tr>
</tbody>
</table>

  written 1234+

• Zoned values _must_ be converted to packed for arithmetic
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

\[
\begin{array}{c}
003^+ \\
+ 003^- \\
000^+
\end{array}
\quad
\begin{array}{c}
500^- \\
500^- \\
000^-
\end{array}
\]

(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
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>
• 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 $\leq 8$ bytes (15 digits) long

• Signs are determined by the rules of algebra

• The Condition Code is unchanged

• Warning: packed decimal products depend on the order of the operands
## Decimal Division

<table>
<thead>
<tr>
<th>Before:</th>
<th>After:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>quotient</td>
</tr>
<tr>
<td>divisor</td>
<td>remainder</td>
</tr>
</tbody>
</table>

- 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
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:

```
<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>
</table>
```

• All instructions process operands from right to left
Test Decimal (TP) Instruction

- TP tests the validity of its operand. Assembler Language syntax:
  \[ \text{TP } D_1(N,B_1) \]

- Its machine instruction format differs from the other instructions:

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

- Valid Assembler Language instruction operand formats:

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

<table>
<thead>
<tr>
<th>Implied Address</th>
<th>S(N)</th>
</tr>
</thead>
</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

  \[ \text{ZAP } D_1(N_1,B_1),D_2(N_2,B_2) \text{ or } \text{Target}(N_1),\text{Source}(N_2) \]

- 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:

  \[ \text{AP} \quad D_1(N_1,B_1),D_2(N_2,B_2) \quad \text{(Same for SP)} \]

- 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

  \[ \text{AP} \quad \text{P123,P9} \quad \text{Result at P123} = 132+. \quad \text{CC}=2 \]

  \[ \text{AP} \quad \text{P9,P234} \quad \text{Result at P9} = 3+, \quad \text{CC}=3 \quad \text{(overflow)} \]

  \[ \text{P123} \quad \text{DC} \quad \text{P’+123’} \]

  \[ \text{P234} \quad \text{DC} \quad \text{P’+234’} \]

  \[ \text{P9} \quad \text{DC} \quad \text{P’+9’} \]
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:

  \[
  \begin{align*}
  \text{CP} &= P' + 5', = P' + 3' & \text{CC} = 2 \\
  \text{CP} &= P' + 3', = P' + 5' & \text{CC} = 1 \\
  \text{CP} &= P' + 0', = P' - 0' & \text{CC} = 0
  \end{align*}
  \]
Multiply Decimal (MP) Instruction

- Assembler Language syntax:

\[
\text{MP } D_1(N_1,B_1),D_2(N_2,B_2)
\]

- Operand length restrictions:

\[
\begin{align*}
2 & \leq N_1 \leq 16, & 1 & \leq N_2 \leq 8, & N_1 & > N_2 \\
1 & \leq L_1 \leq 15, & 0 & \leq L_2 \leq 7, & L_1 & > L_2
\end{align*}
\]

- 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
  bbbbbbbbbbbbbbbbbbbbbbbbs
```

  First operand (multiplicand)

  Second operand (multiplier)
Divide Decimal (DP) Instruction

- **Assembler Language syntax:**
  
  \[
  \text{DP} \quad \text{D}_1(N_1,B_1),\text{D}_2(N_2,B_2)
  \]

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

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>dividend</td>
<td>quotient</td>
</tr>
<tr>
<td>divisor</td>
<td>divisor</td>
</tr>
</tbody>
</table>

- 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:**

  \[
  \begin{align*}
  \text{ZAP} & \quad \text{Dvnd},=\text{P}'162843' & \quad \text{Initialize dividend} \\
  \text{DP} & \quad \text{Dvnd},=\text{P}'762' & \quad \text{Divide by 762} \\
  \text{Dvnd} & \quad \text{DS} \quad \text{PL4} & \quad \text{Result: X'213C537C'}
  \end{align*}
  \]
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:
  
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>L_1</td>
<td>I_3</td>
<td>B_1</td>
<td>D_1</td>
</tr>
<tr>
<td>B_2</td>
<td>D_2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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:
  \[
  \begin{align*}
  \text{SRP } X,3,0 & \quad \text{Multiply operand at } X \text{ by } 1000 \\
  \text{SRP } X,64-3,5 & \quad \text{Divide operand at } X \text{ by } 1000, \text{ round last digit}
  \end{align*}
  \]
- Possible overflow on left shifts
- Rounded results are slightly biased
• MVO moves the second operand to the first, but offset to the left by 4 bits
• Assembler Language syntax:

  \[
  \text{MVO } D_1(N_1, B_1), D_2(N_2, &B2)
  \]

• Example of two 4-byte operands with signs s1, s2:

  ┌───┬───┬───┬───┬───┬───┬───┬───┐
  │ x x │ x x │ x s1 │
  └───┴───┴───┴───┘

  ┌───┬───┬───┬───┬───┬───┬───┬───┐
  │ b c │ d e │ f g │ h s2 │
  └───┴───┴───┴───┘

  ┌───┬───┬───┬───┬───┬───┬───┬───┐
  │ c d │ e f │ g h │ s2 s1 │
  └───┴───┴───┴───┘

• If \( N_2 \geq N_1 \), high-order digits are lost
• If \( N_2 < N_1 \), high-order digits positions are filled with zeros
Before SRP was available, MVO was used for 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.
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:

  \[
  \text{DC P}'123.4' \quad \text{Precision} = 4, \text{Scale} = 1 \quad (\text{Integer attribute} = 3)
  \]

  \[
  \text{DC P}'5.678' \quad \text{Precision} = 4, \text{Scale} = 3 \quad (\text{Integer attribute} = 1)
  \]

- If I = number of integer digits, and F = number of fraction digits, then Precision \(P=I+F\), and Value = I.F

- Given operands 1 and 2 (I₁.F₁ and I₂.F₂), then:
  - Sum or Difference I.F: \(I=\text{Max}(I₁,I₂), F=\text{Max}(F₁,F₂)\)
  - Product I.F: \(I=I₁+I₂, F=F₁+F₂\)
  - Quotient I.F: \(I=I₁+F₂; \text{for an N-digit result, } F=N-I\)

- You must keep these in mind doing scaled packed decimal arithmetic
Section 30: Converting and Formatting Packed Decimal 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
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
    
    ```
    CVD 7, WorkArea  Convert to packed decimal at WorkArea
    CVDY 7, WorkArea  Convert to packed decimal at WorkArea
    WorkArea DS  D  Result = X'00000000 0000135C'
    ```

- Example: suppose $c(GG8) = X'4000000000000000'$ = $2^{62}$
  - CVDG converts 64-bit integers to 16 bytes of 31 packed decimal digits.
    
    ```
    CVDG 8, WrkArea2  Convert to 16-byte packed decimal
    WrkArea2 DS  2D  Result=X'00000000 00004611 68601842 7387904C'
    ```

- The operands need not be aligned on any specific boundary
  - But doubleword alignment can improve performance
• Convert 8- or 16-byte packed decimal data to 2’s complement binary in a register

```
CVB 0,PackNum    Result in GR0 = X’FFFFFF79’ = -135
CVBG 9,PackNum2   Result in GG9 = X’FFFFFF8F779F22087’
```

```
PackNum DC OD,PL8’-135’
PackNum2 DC OD,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 OD,PL8’123456789012345’ Exceeds 2**31 (somewhat)
```

• The operands need not be aligned on any specific boundary
  - Doubleword alignment can improve performance
• Assembler Language syntax for ED and EDMK:

\[ \text{mnem } D_1(N,B_1),D_2(B_2) \text{ or Pattern}(N),\text{PackData} \]

• 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:

```
 d d d d d d d s → edit → C C C 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 actions depend on which pattern character (PC) is being processed, and

- What happened previously, as determined by CPU’s Significance Indicator (SI)

There are five types of pattern characters (PCs):

1. Fill Character (FC), may have any value; the first byte of the pattern

2. Digit Selector (DS), X’20′ (DS notated \(d\))
   - If a nonzero data digit has been processed previously, or the SI is 1, or the current digit is nonzero, it is converted to EBCDIC and the SI is set to 1. Otherwise the DS is replaced by the FC.

3. Digit Selector and Significance Start (SS), X’21′ (SS notated \(s\))
   - The SI is set to 1; if the current digit is nonzero, it is converted to EBCDIC. Otherwise the SS is replaced by the FC.

4. Field Separator (FS), X’22′ (FS notated \(f\))
   - The SI is reset to 0, and the FS is replaced by the FC.

5. Message character having any other value; unchanged or replaced by FC
   - Things like decimal points, currency signs, +/- signs, and text like CREDIT

A pattern like X’402020204B202120′ is represented by C’•ddd,dsd′
Each edit step produces one of three results, in this priority:

1. A zoned source digit replaces a DS or SS in the pattern
   If: the digit is nonzero, or the SI is ON

2. The FC replaces the pattern character
   If: the SI is OFF, or the pattern character is FS

3. The pattern character is unchanged
   If: the SI is ON, or the pattern character is the FC

SI settings:

OFF: (1) at start, (2) after FS, (3) source byte has + code in rightmost digit

ON: if no + code in rightmost digit, then (1) SS and valid digit, (2) DS and nonzero digit

CC settings:

<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>
Simple Examples of Editing

1. A small number

\[
\begin{align*}
\text{MVC} & \quad \text{PgNum}, \text{PgNPat} & \text{Copy pattern to result area} \\
\text{ED} & \quad \text{PgNum}(4), \text{PgNo} & \text{Convert to characters}
\end{align*}
\]

\[
\begin{align*}
\text{PgNo} & \quad \text{DC} \quad \text{PL2}’7’ & \text{Page number 007+} \\
\text{PgNum} & \quad \text{DS} \quad \text{CL4} & \text{Edited result} = \text{C’•••7’} \\
\text{PgNPat} & \quad \text{DC} \quad \text{C’’,3X’20’} & \text{Pattern} = \text{C’ ddd’}
\end{align*}
\]

• A zero value converts to all blanks!

2. 32-bit binary integer; note SS before last DS

\[
\begin{align*}
\text{L} & \quad 0, \text{Num} & \text{Get nonnegative binary number} \\
\text{CVD} & \quad 0, \text{WorkArea} & \text{Convert to packed decimal} \\
\text{MVC} & \quad \text{LineX}, \text{Pat} & \text{Move pattern to print line} \\
\text{ED} & \quad \text{LineX}, \text{WorkArea}+2 & \text{Start edit with high-order digits}
\end{align*}
\]

\[
\begin{align*}
\text{Num} & \quad \text{DC} \quad \text{F’1234567890’} & \text{Number to be printed} \\
\text{WorkArea} & \quad \text{DS} \quad \text{D} & \text{8-byte work area for CVD} \\
\text{Pat} & \quad \text{DC} \quad \text{C’’,9X’20’,X’2120’} & \text{Pattern} = \text{C’•ddddddezzs’d’} \\
\text{LineX} & \quad \text{DS} \quad \text{CL12} & \text{Edited result here, C’••1234567890’}
\end{align*}
\]

• Editing starts after 4 high-order zero digits; the SS ensures that a zero value displays at least one digit
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   OD,XL8  Work area for CVD
Pat    DC    C' ',3X'20206B20',X'2120'  C'•ss,sss,sss,sds' (X'6B' is a comma)
LineX  DS    CL(LineX)  Edited result: C'•1,234,567,890'
```

• Editing negative values (like a credit on a charge-card bill)

```
MVC    LinB,Pat2  Move pattern to print line
ED    LinB,Balance  Edit to printable form
    __ __
Balance DC    P'–0012345'  Credit balance of $123.45
Pat2   DC    C' ',X'20206B2020214B2020',C' CREDIT'  Pattern = C'•dd,dds.dd•CREDIT'
PatX   Equ    *  Used for defining length of Pat2
Line   DC    C' Your account balance is'
LinB   DS    CL(PatX–Pat2)  Space for edited result

- If the balance is -$123.45 (the bank owes you) the result is
  Your•Account•Balance•is••••123.45•CREDIT
- If the balance is +$321.09 (you owe the bank) the result is
  Your•Account•Balance•is••••321.09•••••••
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'20206B2020214B2020'  C'•dd,dds.dd'
Line     DC  C'Pay Exactly' Precedes the 'Amount:' area
LPat     DS  CL(Line–PayPat) Result = C'•••$987.65'
```

- If the first significant digit is forced by a SS, the SI will be ON and GR1 remains unchanged
• One execution of ED/EDMK can edit multiple fields
  - A field separator (FS) (1) sets the SI OFF, and (2) is replaced by the FC

• Example: edit two packed decimal values

```
ED Pat2,PD2 Edit two packed decimal values
  ─ ─ ─
PD2   DC   P′+024′,P′−135′ Two values
Pat2   DC   X′402021204022202120′ C′•dsdf•dsd′
```

  - The result is C′••24••135′

• With EDMK, if more than one nonzero digit forces the SI ON, only the address of the rightmost is placed in GR1
  - The SI will then be OFF if that digit has a + code in the right digit
• These are complex but powerful instructions
• We use some simple macro instructions for input, output, conversion, and display

CONVERTI converts decimal characters in memory to 32- or 64-bit binary integers in a general register

CONVERTO converts 32- or 64-bit binary integers in a general register to decimal characters, or contents of a floating-point register to hexadecimal characters, in memory

DUMPOUT displays the contents of storage in hexadecimal and character formats

PRINTLIN sends a string of characters to a printer file

PRINTOUT displays the contents of registers and of named areas of memory, and/or terminates execution

READCARD reads an 80-byte record from an input file to a specified area of memory

• Each macro calls an entry point in an automatically generated control section
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
• CONVERTI is written

\<\text{nfs}\> \text{CONVERTI <number>,<address>\[,ERR=<address>\] [,STOP=<address>\]}

- The digits starting at the second operand <address> are converted to binary in the general register designated by <number>
  - The first non-blank character must be +, –, or a decimal digit; if not, GR1 is set to the address of the invalid character
  - ERR= specifies an address to receive control for an invalid <number> or a too-large converted value
  - STOP= specifies an address to receive control for an invalid character in the input
  - If either condition occurs and neither ERR= or STOP= is specified, the program terminates with an error message.

• Example:

\text{CONVERTI 3,Data} \quad \text{c(GR3) = X'00000013', c(GR1) = A(Data+3) ('?'')}

\begin{verbatim}
Data DC C'+019?' Input string
\end{verbatim}
The CONVERTO Macro Instruction

- CONVERTO is written

  <nfs> CONVERTO <number>,<address>

- The contents of the register specified by <number> (not the <number> itself!) are converted to N characters in memory starting at <address>

  \[0 \leq \text{<number>} \leq 15: \text{N}=12\]

  \[16 \leq \text{<number>} \leq 31: \text{N}=21\]

  \[32 \leq \text{<number>} \leq 47: \text{N}=20\]

- The first character of the result is always a blank

- If the value of <number> is not between 0 and 47, the macro is ignored

- Converted negative binary values are preceded with a – sign

- Examples (where • represents a blank character):

  CONVERTO 4,GR4Val \hspace{1cm} c(\text{GR4Val}) = •−2147483648

  CONVERTO 22,GG6Val \hspace{1cm} c(\text{GG6Val}) = •−9223372036854775808

  CONVERTO 34,FR2Val \hspace{1cm} c(\text{FR2Val}) = •X‘FEDCBA9876543210’
The DUMPOUT Macro Instruction

- DUMPOUT prints a formatted display of memory (a “dump”)
  
  \[\texttt{<nfs> DUMPOUT <address>[,<address>]}\]

  - 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:
  
  \[\texttt{Dumpout A,B Dump, including bytes from A to B}\]

  - produces something like this:

  *** DUMPOUT requested at Address 01A102, Statement 797, CC=0
  01A000 1B1190EF F00C58F0 F01405EF 00F12802 0001A000 0001A204 F001A002 00000006 *
  01A020 98EFE000 070090EF F03058F0 F03805EF 00F12802 0001A000 8001A22C 0001A026 *q

© IBM 2015
The PRINTLIN Macro Instruction

- PRINTLIN sends up to 121 characters to a print file
  \[<nfs>\ \text{PRINTLIN} \ <address>[,<number>]\]
  - 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:
  \[\text{PrTtl} \ \text{PRINTLIN} \ \text{Title} \ \\
  \ \\
  \text{Title DC CL121\'1Title for Top Line of a Page'}\]
The PRINTOUT Macro Instruction

- PRINTOUT supports 3 types of operand: <name>, <number>, and *

  - <nfs> PRINTOUT [<name>,...][<number>,...]

  - <nfs> PRINTOUT *

    - * terminates execution; it is treated as the last operand
    - a <number> operand between 0 and 47 refers to a register; other values are treated as an address, or ignored
    - a <name> operand causes the named area to be printed; format and length depend on operand attributes

- Examples:

  PrintOut 1,19,32,*  Print GR1, GG3, FPRO, terminate

- Produces output like this:

  *** 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
READCARD reads 80-byte records into your program

\[
\text{<nfs>} \quad \text{READCARD} \quad \langle\text{address}\rangle[,\langle\text{address}\rangle] \quad \text{<nfs>}
\]

- The first operand specifies the location in your program for the record
- If no records remain ("end of file", EOF)
  1. If the second operand is present, control is returned to that location
  2. If the second operand is omitted, the program is terminated with a message

*** Execution terminated by Reader EOF

Example:

\[
\text{GetARec} \quad \text{READCARD} \quad \text{MyRecord,EndFile}
\]

EndFile   ---   Do something about no more records
PRINTOUT and DUMPOUT Headers

- 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,*
Usage Notes

- All the macros must execute in 24-bit addressing mode, AMODE(24), and reside below the 16MB “line”, RMODE(24)

- The instructions generated by the macros are self-modifying, as is the generated “service” control section; programs using these macros are not reenterable

- Most operands of the form <address>, <name>, and <number> are resolved in S-type address constants, so addressability is required when a macro is invoked

- Be careful not to reference areas outside your program

- At most 8 characters of <name> and <d(b)> operands are displayed by PRINTOUT
• This sample program and its listing and output are shown in the text

```
Sample Program

Print Nogen Suppress expansions
IOSamp Csect , Sample Program
Using *,15 Local base register
SR 1,1 Clear card counter
*

PrintOut
Read ReadCard CardOut,EOF Read card until endfile
LA 1,1(0,1) Increment card counter
PrintOut 1 Print the count register
PrintLin Out,LineLen Print a line
ConvertI 2,CardOut Convert a number into GR2
ConvertO 2,OutData Put it in printable form
PrintLin OutData,L'OutData Print the value
B Read Go back and read again
EOF DumpOut IOSamp,Last Dump everything
XGR 3,3 Set GG3 to 0
BCTGR 3,0 Now set GG3 to -1
PrintOut 1,19,32,* Print GR1, GG3, FPRO, terminate

Out DC C'0' Input Record = '' First part of line
CardOut DC CL80' ',C'' Card image here
LineLen Equ *,Out Define line length
OutData DS CL12 Converted characters
Last Equ * Last byte of program
End
```