5. Assembly Language Programming

Table of Contents

5.1 Number Systems

Decimal Number System (base 10)

- Uses 10 digits: 0 – 9
- $125 = 1 * 10^2 + 2 * 10^1 + 5 * 10^0$

Binary Number System (base 2)

- Uses 2 digits: 0 and 1
- $110101 = 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$

Hexadecimal Number System (base 16)

- Uses 16 symbols: 0 – 9, A, B, C, D, E, F
- $19F = 1 * 16^2 + 9 * 16^1 + 15 * 16^0$

The Numbers 0 to 15 in Decimal, Binary & Hex

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D

14	1110	E
15	1111	F

Converting Binary to Hexadecimal

- Starting from the right, divide the 0's and 1's into groups of 4. Pad on the left with 0's, if needed, to form a group of 4. Find the corresponding hex value from the table.
- 11011011100011 = 0011 0110 1110 0011 = 3 6 E 3 = 36E3

Converting Hexadecimal to Binary

- Convert each symbol to its corresponding binary value.
- 2AF = 2 A F = 0010 1010 1111 = 001010101111

Converting Binary or Hexadecimal to Decimal

- Multiply each symbol by the base value raised to a positional power and then add each product.
- $11011 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 27$
- $2AF = 2 * 16^2 + 10 * 16^1 + 15 * 16^0 = 687$

Converting Decimal to Binary or Hexadecimal

- Divide by the base value (2 or 16) until the quotient is 0. If converting to hex, convert the remainders to hex.
- Here is an example converting 415 (in decimal) to hexadecimal:

415 (in decimal) = 19F (in hex)

```
   0
16 |  1      Remainder 1   =>  1
16 | 25      Remainder 9   =>  9
16 |415      Remainder 15  =>  F
```

working upwards

Arithmetic

Binary Addition

```
                          1        11
  0       0       1         1        1
+ 0     + 1     + 0     +   1    +   1
  0       1       1        10       11
```

```
 111111
  11010110
+  1101101
 101000011
```

Binary Subtraction

```
                          02
   0        1        1     ~~10~~
-  0     -  0     -  1     -  1
   0        1        0        1
```

```
    1
   0~~2~~202  02
  1~~1~~00101~~1~~1~~0~~
-   11010001
  1001011101
```

Hexadecimal Addition

```
  A27CB4          39CDF106
+ 6E3095        + A6F278C
 110AD49          443D1892
```

Hexadecimal Subtraction

```
  A52CF3          3B0029
- 2B7169        - 1765A4
  79BB8A          239A85
```

Storage

The main storage of a computer's memory is made up of bits (aka binary digits). Numbers in storage are limited in the number of bits allowed to represent a number. Numbers stored in computer memory are ***always signed numbers***. This means that they have a way to indicate positivity and negativity.

```
1 bit => binary 0 or 1

1 byte => 8 bits => 2 hex digits

1 halfword => 2 bytes => 16 bits => 4 hex digits

1 fullword => 4 bytes => 32 bits => 8 hex digits => 2 halfwords

1 doubleword => 8 bytes => 64 bits => 16 hex digits => 2 full words
```

- Largest positive hexadecimal value that can be stored: 7FFFFFFF or $2^{31} - 1$
- If the first digit is 0 through 7, it is a positive hex number.

- Most negative hexadecimal value that can be stored: `80000000` or -2^{31}

- If the first digit is 8 through F, it is a negative hex number.

Largest positive binary value: `01111111111111111111111111111111`

- First digit is called the **sign bit.**

- If 0, it is a positive binary number.

- If 1, it is a negative binary number.

Negative numbers are stored by taking the two's complement of the absolute value of the number.

To find binary 2's complement:

1. Switch all of the 0s to 1s and 1s to 0s (finding the 1's complement).
2. Add 1.

```
100111100 => 011000011
           +          1
             011000100
```

To find hexadecimal 2's complement:

1. Subtract the number from FFFFFFFF.
2. Add 1.

```
  FFFFFFFF          FFFFFFFF          FFFFFFFF
- 002BCF06        - 00000001        - FFD430FA
  FFD430F9          FFFFFFFE          002BCF05
+        1        +        1        +        1
  FFD430FA          FFFFFFFF          002BCF06
```

If signed numbers, do arithmetic using the two's complement:

- **Addition** – same

- **Subtraction** – Find the two's complement of subtrahend (# after the subtraction sign) and add it to the number

Overflow:

Occurs when a number becomes too large for its representation scheme. In other words, the number you are trying to represent cannot be represented in the maximum bits allowed plus still have a sign bit that is correct.

To check for overflow:

1. Convert the 1st digit of each number to binary.
2. Add the binary values together.
3. If the last two carry bits are the same, no overflow.
4. If they are different, overflow.

```
                  00 ◄── NO overflow (carry in of 0 matches the carry out of 0)
  729B6320     7 => 0111
+ 8A5C973C     8 => 1000
  FCF7FA5C          1111


                  10 ◄── overflow (the carry in of 0 doesn't match carry out of 1)
  92B176C0     9 => 1001
+ 859237A4     8 => 1000
  1843AE64          0001


                                                 01111  <=  overflow
  328AC105                   328AC105    3 =>   0011
- 807B96AF => 7F846951 => + 7F846951    7 => + 0111
                            B20F2A56          1011
```

5.2 D(B) and D(X,B) Addressing

Assembler Identifiers

We are often required to give names to entities in programming. In Assembler language, these names can be between 1 and 8 characters, can contain letters a-z and A-Z, digits 0 through 9 and the national characters @, # and $. A name can begin with either a letter or one of the national characters but do not begin with a national character as they indicate special use by convention.

Today's Assembler accepts mixed-case names, i.e., upper and lower case letters mixed. This may cause problems, though, as the Assembler processes ALL letters as upper case. Thus, a symbol like AbCdEfgh is considered ***to be the same*** as symbol ABCDEFGH.

The recommendation is that only capital letters be used throughout the Assembler language program.

Another Form of Storage:

Along with main storage, there are also 16 General Purpose Registers (GPRs).

- Accessed by the computer faster than main storage
- Hold 32 bits
- Numbered from 0 to 15

Addressing Main Storage:

In our fake ASSIST world, we pretend that main memory is only 16 MB when, in reality, it is much much bigger. Each and every byte of 16 MB can be referenced by its address in 24-bits (3 bytes). This is called 24-bit addressing.

When ESA, or Enterprise Systems Architecture was developed by IBM, available main memory went up to a whopping 2 GB. Each and every byte of 2 GB can be referenced by its address in 31 bits (almost 4 bytes). This is called 31-bit addressing.

Now, with z/OS and its 64-bit addressing capability, we could theoretically address each and every byte up to 16 EB (1 exabyte = 1,000,000 terabytes!). But, we are really only allowed to reference up to 64 GB.

The giant storage capabilities for main memory allow main memory to be logically partitioned thus allowing the look of many computers on one piece of hardware. This works really well for a corporation that might have their one machine logically partitioned into three partitions. F, for example, a development partition in which the programmers and developers work, a quality assurance testing partition where their software is tested by the users, and, finally, a production partition where the real stuff happens. You will learn more about this in CSCI 465/565 – Enterprise Application Environments.

Every byte of storage has an **absolute address**, ranging from `000000` to `FFFFFF`. The absolute address is the address of the first byte of an instruction or data item that is referred to.

A slight problem: a programmer does not know where his/her program will be stored in main storage.

To solve the problem: use **base-displacement, relative** or **explicit addressing**. The idea behind this is that the position of two items is fixed. If the absolute address of one is known, then the address of the other statement can be figured out by calculating the displacement between the two items.

Relative address = base address + displacement

The base address is placed in a GPR when a program is executed. The GPR is called a **base register**.

Two forms of relative addressing:

1. **D(B)**

   - **D** is the displacement. Decimal number between 0 - 4095
   - **B** is the base register number. GPR between 0 - 15

   ```
   4(5) read "4 off of register 5"
   ```

   ```
   36(7)
   ```

   How it works: Convert the displacement to hex and then add it to the LAST three bytes of the address in the register. 1st byte is ignored.

   Assume register 7 holds `010234A0`, then:

   ```
   36(7)  =  010234A0
          +        24
             0234C4
   ```

   Special Case: If the register number is 0, the absolute address `00000000` is used in the calculation.

   If register 0 holds `2A763598`, then `36(0)` results in `000024`.

2. **D(X,B)**

   - **D** is the displacement. Decimal number between 0 - 4095
   - **X** is the index register number. GPR between 0 - 15
   - **B** is the base register number. GPR between 0 - 15

   ```
   4(5,7)
   ```

   `8(9)` -- 9 is the index register

   `2(,3)` -- 3 is the base register

   Register number defaults to 0 if left off.

   Assume register 4 holds `0500029B` and register 7 holds `000027A4`, then:

```
3(4,7)  =  000027A4
           0500029B
         +        3
           002A42
```

5.3 DS and DC Statements

To define empty storage (DS = Define Storage):

```
  Format: label  DS  rSLn

    r   repetition factor
        - number of n length storage classes to repeat
        - optional decimal number
        - default: 1


    Ln  length
        - n is the number of bytes of the storage area
        - optional decimal number
        - L must be coded if n is going to be specified
        - default: depends on the storage class


    S   storage class
        - type of data that will be stored
        - two types

          F  - fullword data
             - numbers to do arithmetic with
             - default length: 4 bytes
             - data type: 32-bit twos complement integer
             - alignment: fullword boundary
                          absolute address divisible by 4

               NUM1  DS  F    ==>  1 fullword  ==> 4 bytes
               NUM2  DS  5F   ==>  5 fullwords  ==> 20 bytes
               NUM3  DS  FL3  ==>  3 bytes
               NUM4  DS  4FL2 ==>  8 bytes


          C  - Character data
             - default length: 1 byte
             - data type: EBCDIC characters (1 character/byte)
             - alignment: none

               TEXT1  DS  CL4   ==>  4 bytes
               TEXT2  DS  10CL6 ==>  60 bytes
               TEXT3  DS  10C   ==>  10 bytes
```

To define a constant (DC = Define Constant):

```
  Format: label  DC  rSLn‘value’

    r   repetition factor
```

```
    S   storage class
    Ln  length

    value   constant (or initial) value
            - enclosed in single quotes
```

```
  In storage:

  NUM1   DC  F’27’        0000001B

  NUM2   DC  F’-1’        FFFFFFFF

  NUM3   DC  F’2,-1,18'   00000002FFFFFFFF00000012

  NUM4   DC  3F’1’        000000010000000100000001

  TEXT1  DC  C’HELLO’     C8C5D3D3D6

  TEXT2  DC  CL8’HELLO’   C8C5D3D3D6404040

  TEXT3  DC  CL3’HELLO’   C8C5D3

  TEXT4  DS  CL4’BYE’     F5F5F5F5  <- empty storage
```

NOTE: a ‘value’ may be specified on a DS statement, but it will be ignored

5.4 Basic Assembler Instructions

Generic Format:

```
          optional_label    mnemonic    operands    line_documentation
Column:   1                 10          16          up to 71
```

Some instructions set a 2 bit **Condition Code (CC)**, which reflects the execution of an instruction.

RX Instructions: R - register, X - D(X,B) address format, 4 bytes

1. **Load**

   ```
   Column:          1     1    (the same in all instructions below)
           1        0     6
   Format: label    L     R,D(X,B)
   ```

   Copies the 4 bytes at the absolute address represented by D(X,B) into R. The previous contents of R are overwritten.

2. **Store**

   ```
   Format: label    ST    R,D(X,B)
   ```

   Stores the contents of R at the absolute address represented by D(X,B).

3. **Add**

   ```
   Format: label    A     R,D(X,B)
   ```

   Takes the 4 bytes from the absolute address represented by D(X,B) and adds it to the contents of R. The result is stored in R.

   Sets the Condition Code.

Code	Meaning
0	Result is equal to 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

4. **Subtract**

   ```
   Format: label    S     R,D(X,B)
   ```

   Takes the 4 bytes from the absolute address represented by D(X,B) and subtracts it from the contents of R. The result is stored in R.

Sets the Condition Code.

Code	Meaning
0	Result is equal to 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

RR Instructions: 2 registers as the operands, 2 bytes

1. **Load Register**

   Format: label LR R_1,R_2

   Copies the contents of R_2 into R_1. The previous contents of R_1 are overwritten.

2. **Add Register**

   Format: label AR R_1,R_2

   Adds the contents of R_2 to R_1. Result is placed in R_1.

   Sets the Condition Code.

Code	Meaning
0	Result is equal to 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

3. **Subtract Register**

   Format: label SR R_1,R_2

   Subtracts the contents of R_2 from R_1. Result is placed in R_1.

   Sets the Condition Code.

Code	Meaning
0	Result is equal to 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

5.5 Sections of an Assembler Program

- **The CSECT statement**

  - Format 1: label CSECT
  - Format 2: label CSECT , comment
  - defines the beginning of a Control SECTion (equivalent to a function, subroutine, or a procedure)

- **The END statement**

  - Format: END label
  - Defines the end of what should be assembled.
  - Last line of an assembler program.
  - label is usually label on the main or 1st CSECT. If specified, then the CSECT named label is the entry point of the program.

- **Last line of EXECUTABLE code**

  - BCR B'1111',14 or BR 14 (using the extended mnemonic version)
  - returns control back to the operating system

Linkage Conventions:

1. On entry to an assembler program, R15 contains the address of the entry point (ie. the beginning of the program). R15 may be used as a base register.

2. On entry to an assembler program, R14 contains the **return address**, which is an address in the operating system to branch to when the program is finished.

A Sample ASSEMBLER Program:

```
Address          Code to execute
 000000          EXMPL1   CSECT
 000000                   L     5,24(,15)
 000004                   A     5,28(,15)
 000008                   ST    5,36(,15)
 00000C                   L     4,32(,15)
 000010                   SR    4,5
 000012                   ST    4,40(,15)
 000016                   BCR   B'1111',14
 000018          NUM1     DC    F'15'
 00001C          NUM2     DC    F'7'
 000020          NUM3     DC    F'8'
 000024          RESULT1  DS    F
 000028          RESULT2  DS    F
                          END   EXMPL1
```

Wouldn’t it have been nice to be able to use the labels NUM1, NUM2, etc...?

Implicit Addressing

- **Implicit address** - an address that the assembler will be required to convert to an explicit address
- Use a label (usually on a DC or DS in storage) rather than a D(B) or D(X,B) address when coding an instruction
- Format 1: label
- Format 2: label±n -- where n is a decimal displacement
- Format 3: label(R) -- where R is the # of the index register
- Format 4: label±n(R) -- where n is decimal displacement, R is index reg
- Must first establish addressibility (supply the assembler with the base address) with the USING statement.
    - Format: USING label,R
    - **R** - register # that contains a valid base address
    - **label** - label in the program that corresponds to base address in R
    - USING does not take up any space.

The Revised Sample ASSEMBLER Program:

```
Address          Code to execute
 000000          EXMPL1   CSECT
 000000                   USING EXMPL1,15
 000000                   L     5,NUM1
 000004                   A     5,NUM2
 000008                   ST    5,RESULT1
 00000C                   L     4,NUM3
 000010                   SR    4,5
 000012                   ST    4,RESULT2
 000016                   BCR   B’1111’,14
 000018          NUM1     DC    F’15’
 00001C          NUM2     DC    F’7’
 000020          NUM3     DC    F’8’
 000024          RESULT1  DS    F
 000028          RESULT2  DS    F
                          END   EXMPL1
```

5.6 Encoding and Decoding Instructions

Assembler developers need to know how to encode and decode instructions.

Assembler instructions consist of an instruction mnemonic in column 10 and the operands for that instruction begin in column 16.

Different instructions require different formats of operands.

For example, as an RR, or register to register, type instruction, the AR (Add Register) instruction takes two register numbers as operands:

```
         AR    4,9       ADD CONTENTS OF R9 TO R4
```

And, in contrast, as an RX, or register to D(X,B), type instruction, the A (Add) instruction takes a register number as its first operand and a D(X,B) address as its second:

```
         A     3,64(0,15)    ADD FULLWORD AT 64 OFF OF R15 TO R3
```

Some of the most common RR instructions are:

`AR` Add Register
`SR` Subtract Register
`LR` Load Register

And some of the most common RX instructions are:

`L` Load Register
`ST` Store Register
`A` Add
`S` Subtract
`LA` Load Address

Machine instruction formats are provided in the System/370 Reference Summary, or "Yellow Card".

Listed there are, among others, a bitwise "map" or "layout" of both the RR instruction and the RX instruction.

Use these as references!

Encoding Instructions

RR instructions are always 2 bytes long.

RR instructions always have the format:

`bits 0-7` are the instruction's operation code, or "op code", in hexadecimal
`bits 8-11` represent the first operand's register number in hexadecimal
`bits 12-15` represent the second operand's register number in hexadecimal

For example:

`AR`	`5,6`	would be encoded in hexadecimal as	`1A56`
`LR`	`7,13`	would be encoded in hexidecimal as	`187D`

For the first of these two examples, note that hexadecimal `1A` – found under `AR` on page 4 of the Yellow Card – is the operation code for the Add Register instruction, `5` is the number of the first operand register and `6` is the number of the second operand register.

For the second of these two examples, hexadecimal `18` – found under `LR` on page 5 of the Yellow Card – is the operation code for the Load Register instruction, `7` is the number of the first operand register and hexadecimal `D`, or 13 in decimal, is the number of the second operand register number.

RX instructions are always 4 bytes long because there is more information to store in the encoded instruction.

RX instructions always have the format:

`bits 0-7`	are the instruction's operation code, or "op code", in hexadecimal and is two hexadecimal digits
`bits 8-11`	represent the first operand's register number in hexadecimal and is one hexadecimal digit
`bits 12-15`	represent the index register's number in hexadecimal (if no index register is specified, it defaults to 0) and is one hexadecimal digit
`bits 16-19`	represent the base register's number in hexadecimal (if no base register is specified, it defaults to 0) and is one hexadecimal digit
`bits 20-31`	represent the displacement in bytes in hexadecimal off of the base register

For example:

`ST`	`4,35(2,5)`	would be encoded in hexadecimal as	`50425023`
`L`	`12,50(0,7)`	would be encoded in hexadecimal as	`58C07032`

Special examples:

`L`	`12,50(,7)`	would also be encoded in hexadecimal as	`58C07032`

but

`L`	`12,50(7)`	would be encoded in hexadecimal as	`58C70032`

Note that the register numbers inside the set of parentheses are "positional" with the register number listed before the comma indicating the index register and the one following the comma indicating the base register.

In the first special example above, the `(,7)` indicates that register 7 is the base register and the unlisted index register defaults to register 0.

In the second special example above, the `(7)` indicates that register 7 is the index register, there is no comma and the unlisted base register defaults to register 0.

Extra special example:

```
LA      2,258           would be encoded in hexadecimal as    41200102
```

This is equivalent to coding:

```
LA      2,258(0,0)      which would be encoded in hexadecimal
```

Using the Load Address instruction in this form is the most efficient way to put an integer value between 1 and 4095, inclusive, into a register.

When resolving addresses, remember that register 0's contents are ignored.

Therefore,

```
LA      9,355
```

will put the decimal value 355 in binary in register 9.

This method is far better than:

```
L       9,=F'355'
```

which has the same result but takes an additional fullword of storage for the fullword literal.

Decoding

Decoding an instruction is simply the reverse of the above.

The first digit of the two hexadecimal digit operation code determines the length of the encoded instruction.

Those operation codes beginning with hexadecimal 0 through 3, inclusive, are two-byte instructions.

Those beginning with hexadecimal 4 through B, inclusive, are four-byte instructions.

Those beginning with hexadecimal C through F, inclusive, are six-byte instructions.

5.7 Compare Instructions and Branching

Compare Instructions

RX Format: `label  C  R,D(X,B)`

- Compares the fullword in `R` with the fullword at `D(X,B)`

Sets the Condition Code

Code	Meaning
0	Equality
1	Fullword in `R` is less than fullword at `D(X,B)`
2	Fullword in `R` is greater than fullword at `D(X,B)`

`RR` Format: `label  CR  R1,R2`

- Compares the value in R_1 to the value in R_2

Sets the Condition Code

Code	Meaning
0	Equality
1	Contents of R_1 is less than the contents of R_2
2	Contents of R_1 is greater than the contents of R_2

Branching

Conditional Branch

- used to alter the flow of program execution depending on the condition code set by an instruction

RR Format: `label  BCR  B'mask',R`

- **mask** is a 4 bit binary mask indicating which condition code(s) to branch on

- **R** is the register with the address to which to branch

RX Format: `label  BC  B'mask',D(X,B)`

- **mask** is a 4 bit binary mask indicating which condition code(s) to branch on

- **D(X,B)** is the address to which to branch

`B'mask'  ==>  B'bbbb'==>  B'0123'`

where *0123* represent the possible condition codes

If `b` is `1`, branch to the address in `R` on this condition code.

If b is 0, do not branch.

Examples

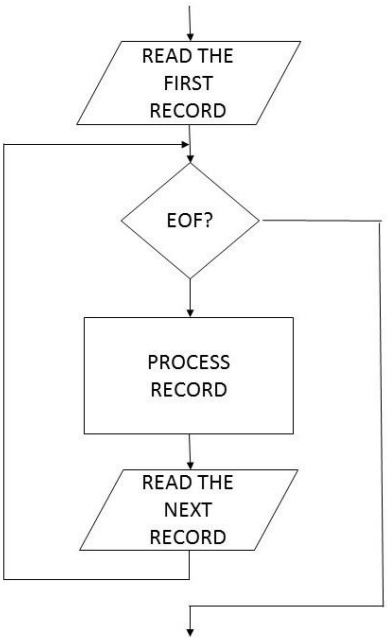
```
        AR      3,4
        BC      B'1000',HERE    branch to label HERE if result of the addition is 0

        SR      3,4
        BC      B'1100',THERE   branch to label THERE if result is equal to 0 or less than 0
```

NOTE: Using the mask B'1111', makes an unconditional branch (it will always branch)

Example Read Loop

```
        XREAD BUFFER,80              read first record in the file
LOOP1   BC    B'0100',ENDLOOP1      branch to ENDLOOP1 if end-of-file is reached
        .
        .                           body of the loop: process the record
        .
        XREAD BUFFER,80             read the next record in the file
        BC    B'1111',LOOP1         branch back to LOOP to check for end-of-file
ENDLOOP1 and go on…                 label for continuation after reading all of the records
```

The following is the flowchart of a structured read loop used to read a file (or any input) that has an unknown number of records or is empty.



Unconditional Branch

Branches to a specific address no matter what the condition code is.

RX Format: `B  D(X,B)`

- Branches to `D(X,B)`

RR Format: `BR  R`

- Branches to the address in `R`

Now, with the condition code and branching, we can write decision structures (IFs) and repetition structures (LOOPs).

5.8 Assembler Literals

Assembler literals allow you to put a hardcoded value within a program rather than having to use a DC in storage.

Format: `=S'value'`

- S is a storage class for a `DC (F or C)`

```
=F'4'   =C'Hello'
```

If literals are used in a program, code `LTORG` directly after the

```
BCR   B'1111',14  or  BR    14
```

line.

Literals can be substituted for a D(X,B) address in instructions

- `L  5,=F'4'` puts 4 into register 5
- `C  3,FOUR  ==>  C     3,=F'4'`

5.9 Multiplication and Division

Multiplication

- Multiply two 32-bit numbers to obtain a 64-bit result that is spread between an even-odd pair of registers
- **RX Format:** label M R,D(X,B)
    - **R** is an even numbered register representing an even-odd pair of registers. The number to multiply must be in the odd numbered register
    - **D(X,B)** is the address of a fullword number to multiply by
- **RR Format:** label MR R_1,R_2
    - **R_1** is an even numbered register representing an even-odd pair of registers. The number to multiply must be in the odd numbered register
    - **R_2** is the number of a register containing the number to multiply by

Example 1: M 4,TWO where TWO DC F'2'

BEFORE: R4 = 05002037 R5 = FFFFFFFD (-3 in decimal)

AFTER: R4 = FFFFFFFF R5 = FFFFFFFA

This is the 2-register, or 64-bit, representation of -6

Example 2: MR 2,1

BEFORE: R1 = 00000004 R2 = 0000FFFF R3 = 00000005

AFTER: R1 = 00000004 R2 = 00000000 R3 = 00000014

This is the 2-register, or 64-bit, representation of 20

Example 3: MR 2,3 => squares the value in register 3

Division

- Divide a 64-bit number that is stored between an even-odd pair of registers by a 32-bit number.
- Results in a 32-bit quotient stored in the odd register and a 32-bit remainder stored in the even register.
- The remainder will ALWAYS have the same sign as the number being divided.
- Could cause a S0C9 - Fixed Point Divide Exception, which is usually caused by division by 0
- **RX Format:** label D R,D(X,B)
    - **R** is an even numbered register representing an even-odd pair of registers. The dividend (ie. the number to be divided) must be spread between the two registers
    - **D(X,B)** is the address of a fullword number to divide by (ie. the divisor)
- **RR Format:** label DR R_1,R_2
    - **R_1** is an even numbered register representing an even-odd pair of registers. The dividend (ie. the number to be divided) must be spread between the two registers
    - **R_2** is the number of a register containing the number to divide by (ie. the divisor)

EXAMPLE 1: DR 2,8

Before: R2 = 00000000 R3 = 00000007 R8 = FFFFFFFE (-2 in decimal)

After: R2 = 00000001 R3 = FFFFFFFD R8 = FFFFFFFE
  REMAINDER QUOTIENT (-3 in decimal)

EXAMPLE 2: D 4,=F'2'

Before: R4 = 00000A00 R5 = 00000007

After: Error (SOC 9) because the quotient will not be 32 bits

To get the desired results:

```
M     4,=F'1'   to zero out the even register
D     4,=F'2'

After:     R4 = 00000001    R5 = 00000003
```

5.10 More Load Instructions

Load and Test Register

Format: LTR R_1,R_2

- loads the contents of R_2 into R_1 and sets only sets the condition code. Of course, R_1's contents are lost but the register you are testing, R_2, with the instruction remains unaffected.

Code	Meaning
0	R_1 contains 0
1	R_1 contains a negative number
2	R_1 contains a positive number

- If you do not want to lose the contents of R_1, you can still use this instruction to check if a register contains the value 0, a negative number, or a positive number by executing a LTR on a register with itself, such as:

```
LTR   2,2
```

This will not affect the contents of register 2 but WILL set the condition code so that you know what type of signed number is in register 2.

Load Positive Register

Format: LPR R_1,R_2

- loads the absolute value of the contents of R_2 into R_1 and sets the condition code

Code	Meaning
0	R_1 contains 0
1	---
2	R_1 contains a positive number
3	Overflow occurred

- overflow will occur if R_2 contains the maximum negative

Load Negative Register

Format: LNR R_1,R_2

- loads the negative of the absolute value of the contents of R_2

into R_1 and sets the condition code

Code	Meaning
0	R_1 contains 0
1	R_1 contains a negative number

Load Complement Register

Format: LCR R_1,R_2

- loads the complement (opposite) of the contents of R_2 into R_1 and sets the condition code

Code	Meaning
0	R_1 contains 0
1	R_1 contains a negative number
2	R_1 contains a positive number
3	Overflow occurred

- overflow will occur if R_2 contains the maximum negative

Load Address

Format: label LA R,D(X,B)

- We have saved the BEST of the load instructions for last!

- Certainly one of the most important and most useful of ALL Assembler instructions!

- Calculates the absolute address represented by D(X,B) and loads it into the last 3 bytes of R. The first byte is set to 00.

Some Possible Uses:

1. Initialize a register to a value between 0 and 4095

```
LA    10,50     puts the number 50 into register 10
```

Note that this encodes the instruction with a 0 index register and a 0 base register and is equivalent to the instruction:

```
LA    10,50(0,0)
```

2. Add a value from 1 to 4095 to a register

```
     LA    5,4(,5)   adds 4 to the value in register 5 and stores it
                     back into register 5
```

3. Add 3 numbers together

```
     LA    10,375(4,5)   adds 375 to the values in registers 4 and 5
                         and stores it in register 10
```

4. Put the address of a storage field in a register

```
     LA    5,TABLE    puts the address of label TABLE in register 5
```

5.11 EQUates and Extended Mnemonics

EQUATES: Assigning a value to a label

- Format: label EQU expression
- gives label the value of the expression
- every occurrence of label will be treated as if it was the expression

```
R0    EQU   0
R1    EQU   1
...
R15   EQU   15
```

Then R# can be used whenever a register number is needed.

L 3,NUM1 can be replaced with L R3,NUM1

Extended Branch Mnemonics

- intended to make the coding of branch instructions easier by eliminating the need to explicitly specify a mask

After COMPARE instructions:

Extended Mnemonic		Meaning	Replaces	
RX version	RR version		RX version	RR version
BH D(X,B)	BHR R	Branch on HIGH	BC B‘0010’,D(X,B)	BCR B‘0010’,R
BL D(X,B)	BLR R	Branch on LOW	BC B‘0100’,D(X,B)	BCR B‘0100’,R
BE D(X,B)	BER R	Branch on EQUAL	BC B‘1000’,D(X,B)	BCR B‘1000’,R
BNH D(X,B)	BNHR R	Branch on NOT HIGH	BC B‘1101’,D(X,B)	BCR B‘1101’,R
BNL D(X,B)	BNLR R	Branch on NOT LOW	BC B‘1011’,D(X,B)	BCR B‘1011’,R
BNE D(X,B)	BNER R	Branch on NOT EQUAL	BC B‘0111’,D(X,B)	BCR B‘0111’,R

After ARITHMETIC instructions:

Extended Mnemonic		Meaning	Replaces	
RX version	RR version		RX version	RR version
BP D(X,B)	BPR R	Branch on PLUS	BC B‘0010’,D(X,B)	BCR B‘0010’,R
BM D(X,B)	BMR R	Branch on MINUS	BC B‘0100’,D(X,B)	BCR B‘0100’,R
BZ D(X,B)	BZR R	Branch on ZERO	BC B‘1000’,D(X,B)	BCR B‘1000’,R
BO D(X,B)	BOR R	Branch on OVERFLOW	BC B‘0001’,D(X,B)	BCR B‘0001’,R
BNP D(X,B)	BNPR R	Branch on NOT PLUS	BC B‘1101’,D(X,B)	BCR B‘1101’,R
BNM D(X,B)	BNMR R	Branch on NOT MINUS	BC B‘1011’,D(X,B)	BCR B‘1011’,R
BNZ D(X,B)	BNZR R	Branch on NOT ZERO	BC B‘0111’,D(X,B)	BCR B‘0111’,R
BNO D(X,B)	BNOR R	Branch on NOT OVERFLOW	BC B‘1110’,D(X,B)	BCR B‘1110’,R

5.12 BCT and BCTR Branching Instructions

Branch on Count

- RX Format: label BCT R,D(X,B)
    - Decrements the value in R by 1 and branches to D(X,B) if the value in R is not 0
    - If R contains 0, instruction immediately following BCT is execute

Branch on Count Register

- RR Format: label BCTR R_1,R_2
    - Decrements the value in R_1 by 1 and branches to the address in R_2 if the value in R_1 is not 0
    - If R_1 contains 0, instruction immediately following BCTR is executed
    - **EXCEPTION:** If R_2 is register 0, then 1 is subtracted from the value in R_1 and NO branch is taken

Three Versions Looping Examples with Same Result

```
       LA    R4,3              LA    R4,3              LA    R4,3
DO1    LTR   R4,R4      DO1    LTR   R4,R4       DO1   XREAD BUFFER,80
       BZ    ENDDO1            BZ    ENDDO1            XPRNT BUFFER,80
       XREAD BUFFER,80         XREAD BUFFER,80         BCT   R4,DO1
       XPRNT BUFFER,80         XPRNT BUFFER,80   EXIT  BR    R14
       S     R4,=F‘1’          BCTR  R4,R0
       B     DO1               B     DO1
ENDDO1 DS    0H         ENDDO1 DS    0H
EXIT   BR    R14        EXIT   BR    R14
```

CSCI 360 MVC and CLC Instructions

SS Instructions (Storage to Storage Instructions)

- have 2 D(B) operands
- 6 byte instructions
- 3 encoding formats

Move Character

- Format: label MVC $D_1(L,B_1),D_2(B_2)$
- Moves L bytes from $D_2(B_2)$ to $D_1(B_1)$
- L is decimal number between 0 and 256

```
MVC 0(8,R10),0(R4) ==> moves 8 bytes from 0(R4) to 0(R10)

MVC NUM1(6),BUFFER ==> moves 6 bytes from BUFFER to NUM1
```

Compare Logical Character

- Format: label CLC $D_1(L,B_1),D_2(B_2)$
- Compares L EBCDIC bytes at $D_1(B_1)$ with L EBCDIC bytes at $D_2(B_2)$ one byte at a time
- L is decimal number between 0 and 256
- Should be used for characters
- Sets Condition Code

Code	Meaning
0	Equality
1	first operand < second operand
2	first operand > second operand

```
NUM1 DC F'10'
NUM2 DC F'5'

L R3,NUM1
C R3,NUM2

RESULT: NUM1 is greater than NUM2

CLC NUM1(4),NUM2

RESULT: NUM1 is less than NUM2.

Will work in this example since 0A is greater than 05

NUM1 00 00 00 0A
NUM2 00 00 00 05

What if NUM2 DC F'-5'?

RESULT: will not work because 00 is less than FF
```

```
NUM1 00 00 00 0A
NUM2 FF FF FF FB
```

5.13 Immediate Byte Instructions

SI Instructions

Instructions that involve a byte in storage represented by a D(B) address and an immediate byte. The immediate byte is part of the encoded instruction. In fact, it is the second byte of the encoded instruction.

The immediate byte can be specified in 4 ways:

1. Character Immediate Byte

   - enclose the character in single quotes and precede it with the letter `C`

   - Example: `C'$'`

2. Hexadecimal Immediate Byte

   - enclose TWO hexadecimal numbers in single quotes and precede it with the letter `X`

   - Example: `X'5B'`

3. Binary Immediate Byte

   - enclose 8 bits in single quotes and precede it with the letter `B`

   - Example: `B'01011011'`

4. Decimal Immediate Byte

   - code a decimal value between `0` and `255` (it will be converted to hexadecimal when encoded

   - Example: `91`

Move Immediate

Format: `label  MVI  D(B),byte`

- Moves the immediate byte specified by **byte** to D(B)

- Examples

  Character: `MVI  42(R5),C'*'` moves a `*` to the address `42(5)`

  Hexadecimal: `MVI  42(R5),X'5B'` moves a `$` to the address `42(5)`

  Binary: `MVI  4(R5),B'01000000'` moves a space to the address `4(5)`

  Decimal: `MVI  0(R10),80` moves a `&` to the address `0(10)`

Compare Logical Immediate

Format: `label  CLI  D(B),byte`

- Compares the byte at `D(B)` with the immediate byte specified by **`byte`**

- Sets Condition code

Code	Meaning
0	Equality
1	byte at D(B) < immediate byte
2	byte at D(B) > immediate byte

Character: `CLI  4(R7),C'A'` compares the letter `A` with the byte at address `4(7)`

Hexadecimal: `CLI  5(R5),X'F0'` compares the character `0` with the byte at address `5(5)`

Binary: `CLI  4(R5),B'11011000'` compares the letter `Q` with the byte at address `4(5)`

Decimal: `CLI  0(R10),64` compares a space with the byte at address `0(10)`

5.14 STM and LM Instructions

RS Instructions
Instructions that require three operands: 2 registers and a D(B) address

Store Multiple

- Format: label STM R_1,R_2,D(B)
- stores all registers from R_1 through R_2 in contiguous fullwords starting at D(B)
- if R_2 is less than R_1, then R_1 through register 15 and register 0 through R_2 are stored

```
STM   R4,R7,SAVE => stores registers 4, 5, 6, and 7 in 4 fullwords
                    starting at SAVE

STM   R14,R2,SAVE2 => stores registers 14, 15, 0, 1, and 2 in 5
                      fullwords starting at SAVE2
```

Load Multiple

- Format: label LM R_1,R_2,D(B)
- loads all registers from R_1 through R_2 from contiguous fullwords starting at D(B)
- if R_2 is less than R_1, then R_1 through register 15 and register 0 through R_2 are loaded

```
LM   R4,R7,SAVE => loads registers 4, 5, 6, and 7 from 4 fullwords
                   starting at SAVE

LM   R14,R2,SAVE2 => loads registers 14, 15, 0, 1, and 2 from 5
                     fullwords starting at SAVE2
```

5.15 Internal Subroutines

Subroutines

Two Types:

1. Internal Subroutine
    - subroutine that is located within the same CSECT as the calling routine
2. External Subroutine – Covered Later
    - Subroutine that is located outside of the calling routine
    - A separate CSECT

Internal Subroutines

On entrance to a subroutine, register 1 holds the address of a parameter list (if the subroutine gets passed any parameters)

On entrance to a subroutine, the initial value in any register that will be altered by the subroutine should be saved

Before exiting a subroutine, the initial value in any register that was altered by the subroutine should be restored

Subroutine Name: `rtnName  DS    0H`

Setting up a parameter list:

- must be a set of contiguous fullwords, each of which contains the address of a parameter to be passed
- use address constants (adcons): label DC A(expr)
    - if expr is a non-negative integer, the generated fullword will contain the binary representation of the integer (same as doing F‘expr’)
    - if expr is a label or label±n, the generated fullword will contain the address of label or address of label±n

      DC A(5) => 00000005

```
000100 SAVE DS F

DC A(SAVE) => 00000100
```

- expr may be several non-negative integers or label separated by commas

```
DC A(5,SAVE) => 0000000500000100
```

A sample parameter list for a program with internal subroutines:

```
PARMLIST   DC   A(TABLE)
           DC   A(EOT)
```

To call an internal subroutine:

1. If the subroutine requires passed in parameters, put the address of the parameter list into register 1
2. Use the Branch and Link Instruction
    - RX Format: label BAL R,D(X,B)
        - **LINK:** The address of the instruction immediately following the BAL instruction is placed in R (where does it get this address from?!)
        - **BRANCH:** Branches to D(X,B), which is usually rtnName

```
LA    1,PARMLIST
BAL   11,RTNNAME
```

To exit an internal subroutine:

Branch to the instruction immediately following the BAL instruction by executing a BR instruction with the register used in the calling routine BAL instruction

```
BR    11
```

Example stripped down program with one internal subroutine:

```
MAIN     CSECT
         USING MAIN,R15
         STM   R0,R15,MAINSAVE
         ...
```

```
         BAL   R11,SUBRTN
         ...
         LM    R0,R15,MAINSAVE
         BR    R14
         LTORG
**** Storage for MAIN starts here ***
MAINSAVE DS    16F
         ...
*** Storage for MAIN ends here ***
SUBRTN   DS    0H                 SUBRTN starts here
         STM   R0,R15,SUBSAVE
         ...
         LM    R0,R15,SUBSAVE
         BR    R11
         LTORG
*** Storage for SUBRTN starts here ***
SUBSAVE  DS    16F
         ...
*** Storage for SUBRTN ends here ***
         END   MAIN
```

5.16 External Subprograms

External subrprograms are those that are created and maintained separately from the program that will be calling them.

Each routine should begin with:

```
      rtnName  CSECT
```

where rtnName is the label by which the subroutine will be called.

Pattern for a program with external subprograms using ASSIST:

```
      MAINPGM  CSECT
      *
      *
      *** MAINPGM code, LTORG, and storage goes here
      *
      *
      SUBRTN1  CSECT
      *
      *
      *** SUBRTN1 code, LTORG, and storage goes here
      *
      *
      SUBRTN2  CSECT
      *
      *
      *** SUBRTN2 code, LTORG, and storage goes here
      *
      *
               END   MAINPGM
```

Each new control section will automatically begin on a doubleword boundary

LTORG is included within each CSECT to ensure that the literals for each routine are assembled as part of that control section, rather than as a part of the first control section (in the above case MAINPGM).

To call an external subroutine, you must:

1. point register 1 to a parameter list (if needed)
2. get the address of the subroutine into register 15
3. call the routine

Step 2 is accomplished by using a V-type literal:

```
      L     R15,=V(SUBRTN)
```

Step 3 is accomplished by using the BALR instruction

```
      Format: label     BALR  R1,R2
```

The BALR instruction works just like the BAL instruction. R_1 is the address of the instruction immediately following the BALR instruction. R_2 is the address of the routine being called.

Standard Linkage Conventions

1. When control is passed to an external subroutine, register 15 should contain the address of the subroutine, register 14 should contain the address of the next instruction to execute, and register 13 should contain the address of an 18F savearea in the calling routine in which its initial register values will be saved.

2. Parameters are passed to an external subroutine through a parameter list. The address of the parameter list is contained in register 1.

3. Return codes are passed back to a calling routine in register 15.

4. Calculated values are passed back to a calling routine in register 0.

5. The subroutine is responsible for storing the contents of the registers upon entry to the routine and restoring those values before returning control to the calling routine.

Standard Entry Linkage

The following code should be included as the first lines of each CSECT, including the MAIN one:

```
      rtnName  CSECT
               STM   14,12,12(13)
               LR    12,15
               USING rtnName,12
               LA    14,name_of_18F_save_area_in_rtnName
               ST    13,4(,14)
               ST    14,8(,13)
               LR    13,14
```

STM 14,12,12(13) saves all of the calling routine's registers, except for register 13, in the calling routine.

LR 12,15 puts the address of rtnName in register 12.

USING rtnName,12 sets register 12 as the base register for rtnName.

LA 14,name_of_18F_save_area_in_rtnName points register 14 to an area in rtnName where its registers will be saved if rtnName calls another routine.

ST 13,4(,14) stores the address of the calling routine's save area in rtnName's save area. The value in register 13 is known as the **backward pointer**.

ST 14,8(,13) stores the address of rtnName's save area in the calling routine's save area. The value in register 14 is known as the**forward pointer**.

LR 13,14 points register 13 to rtnName's save area in case another routine is going to be called.

Format of the 18F register save area

Unused

Backword Pointer

Forward Pointer

Register 14

Register 15

Register 0

Register 1

Register 2

Register 3

Register 4

Register 5

Register 6

Register 7

Register 8

Register 9

Register 10

Register 11

Register 12

Standard Exit Linkage

The following code should be included as the last lines of executable code in each CSECT if no values are being passed back, including the MAIN one:

```
L     13,4(,13)
LM    14,12,12(13)
BR    14
```

L 13,4(,13) loads the address of the calling routine's save area into register 13.

LM 14,12,12(13) reloads all of the calling routine's registers, except for register 13.

BR 14 returns control to the calling routine.

Exit Linkage for a routine passing a return code back in register 15

All of the registers, except for 13 and 15, are going to be restored.

```
L     13,4(,13)
L     14,12(,13)
LM    0,12,20(13)
BR    14
```

L 13,4(,13) loads the address of the calling routine's save area into register 13.

L 14,12(,13) restores the calling routine's register 14.

LM 0,12,20(13) restores registers 0 through 12 of the calling routine.

BR 14 returns control to the calling routine.

Exit Linkage for a routine passing a calculated value back in reg 0

All of the registers, except for 0 and 13, are going to be restored.

```
L     13,4(,13)
LM    14,15,12(13)
LM    1,12,24(13)
BR    14
```

L 13,4(,13) loads the address of the calling routine's save area into register 13.

LM 14,15,12(13) restores the calling routine's register 14 and 15.

LM 1,12,24(13) restores registers 1 through 12 of the calling routine.

BR 14 returns control to the calling routine.

5.17 Number Types and Decimal Numbers

There are three ways to store numbers on the mainframe. They are:

1. Binary
2. Zoned Decimal
3. Packed Decimal

We have already seen how binary numbers look in storage. Let now look at the other types.

There are two formats for decimal numbers:

1. Zoned Decimal
2. Packed Decimal

Zoned Decimal Numbers

- Generally used for input/output

- Represent one decimal digit per byte

- Each byte of is made up of two hex digits

```
    left  ->  zone digit       right ->  numeric digit
```

- The numeric digit is the number represented by the byte

- The zone digit of all but the RIGHTMOST byte must be F

- The zone digit of the rightmost byte is used to indicate the sign of the number

```
    F, A, C, or E    positive number (just remember the word "FACE")
    B or D           negative number

    F1F2F3F4   zoned decimal representation of +1234

    F1F2F3B4   zoned decimal representation of -1234
```

- Can be generated on a DC statement by using a storage class of Z

    - the initial value can contain a sign or a decimal point

    - if the sign is omitted, assumed to be positive

- if plus sign is specified, rightmost zone digit is C

- if negative sign is specified, rightmost zone digit is D

- if a decimal point is specified, treated as an implied decimal point

```
DC   2Z'-1'         D1D1
DC   Z'+2'          C2
DC   Z'2'           C2
DC   ZL3'4'         F0F0C4
DC   2ZL2'-9'       F0D9F0D9
DC   Z'1.10'        F1F1C0
```

- Maximum zoned decimal number can have 16 digits

Packed Decimal Numbers

- Used for arithmetic

- Each byte, except for the rightmost, represents two decimal digits

- The rightmost byte contains a decimal digit and the sign digit

```
   left  ->  numeric digit      right ->  sign digit
```

- The sign digit is used to indicate the sign of the number

```
F, A, C, or E    positive number
B or D           negative number
```

- To represent a number:

1. Move the sign to the right (end) of the number
2. If there is an even number of digits, add a zero on the left
3. Change the sign to an appropriate sign digit

To represent -1234:

1. 1234-
2. 01234-
3. 01 23 4D

- Can be generated on a DC statement by using a storage class of P

  - the initial value can contain a sign or a decimal point

    - if the sign is omitted, assumed to be positive

    - if plus sign is specified, rightmost sign digit is C

    - if negative sign is specified, rightmost sign digit is D

    - if a decimal point is specified, treated as an implied
      decimal point

```
   DC   2P'-1'         1D1D
   DC   P'+2'          2C
   DC   P'2'           2C
   DC   PL3'4'         00004C
   DC   2PL2'-9'       009D009D
   DC   P'1.10'        110C
```

- Maximum packed decimal number can have 31 digits

5.18 Packed Decimal Instructions

Please note that these are often wrongly considered to be the most difficult of all Assembler instructions, even by advanced programmers. The misconception most likely comes down to the fact that the material was inadequately taught! There is a lot of detail so pay attention!

Declaring packed decimal fields

```
label    DS    PL3      AN UNITIALIZED 3-BYTE PACKED DECIMAL FIELD

label    DC    PL5'0'   AN INITIALIZED 5-BYTE PACKED DECIMAL FIELD
```

In this class, please declare ALL packed decimal fields with DC and initialize it to 0 if nothing else!

To convert from zoned to packed

Format: label PACK $D_1(L_1,B_1),D_2(L_2,B_2)$

- Packs the L_2 byte zoned decimal number at $D_2(B_2)$ and stores it as a L_1 byte packed decimal number at $D_1(B_1)$

- The packing process:

    1. The zone and numeric digit of the **rightmost** byte of the zoned decimal number are reversed and placed in the rightmost byte of the packed decimal number

    2. The remaining numeric digits are moved to the packed decimal number, proceeding from **right to left**. If the zoned number has more digits than the packed number can hold, the extras are ignored. If the zoned number has less digits than the packed number, the remaining packed digits are filled with zero

- The second operand does NOT have to be a valid zoned decimal number

Suppose FLD1 is 4 packed bytes and FLD2 is 7 zoned bytes, with the initial values:

```
  FLD1     00 00 00 00
  FLD2     F9 F8 F7 F6 F5 F4 F3
```

Execution of:

```
  PACK  FLD1(4),FLD2(7)         will set FLD1 to    98 76 54 3F
```

```
  PACK  FLD1(3),FLD2(7)         will set FLD1 to    76 54 3F 00

  PACK  FLD1(4),FLD2(5)         will set FLD1 to    00 98 76 5F
```

To swap the hexadecimal digits of any byte, pack the byte into itself:

```
  PACK  BYTE(1),BYTE(1)         will simply reverse the zone and
                                numeric digit
```

To convert from packed to zoned

Format: label UNPK $D_1(L_1,B_1),D_2(L_2,B_2)$

- Unpacks the L_2 byte packed decimal number at $D_2(B_2)$ and stores it as a L_1 byte zoned decimal number at $D_1(B_1)$

- The unpacking process:

  1. The zone and numeric digit of the rightmost byte of the packed decimal number are reversed and placed in the rightmost byte of the zoned decimal number

  2. The remaining numeric digits are padded with a zone digit of F and moved to the zoned decimal number. If the packed number has more digits than the zoned number can hold, the extras are ignored. If the packed number has less digits than the zoned number, the remaining digits are filled with F0

- The second operand does NOT have to be a valid packed decimal number

Suppose FLD1 is 5 zoned bytes and FLD2 is 3 packed bytes

```
  FLD1        00 00 00 00 00
  FLD2        12 34 5C
```

Execution of:

```
  UNPK  FLD1(5),FLD2(3)        will set FLD1 to     F1 F2 F3 F4 C5

  UNPK  FLD1(1),FLD2(3)        will set FLD1 to     C5 00 00 00 00

  UNPK  FLD1(5),FLD2(2)        will set FLD1 to     F0 F0 F1 F2 43
```

Add Packed

Format: label AP $D_1(L_1,B_1),D_2(L_2,B_2)$

- The L_2 byte packed decimal number at $D_2(B_2)$ is added to the L_1 byte packed decimal number at $D_1(B_1)$. The sum is stored at $D_1(B_1)$.

- If L_1 bytes is not large enough to hold all of the non-zero digits of the result, overflow will occur.

- If either field is not a valid packed decimal number, a data exception (SOC 7) will occur.

- Sets the condition code

Code	Meaning
0	Result is 0
1	Result is negative
2	Result is positive
3	Overflow

Subtract Packed

Format: label SP $D_1(L_1,B_1),D_2(L_2,B_2)$

- The L_2 byte packed decimal number at $D_2(B_2)$ is subtracted from the L_1 byte packed decimal number at $D_1(B_1)$. The difference is stored at $D_1(B_1)$.

- If L_1 bytes is not large enough to hold all of the non-zero digits of the result, overflow will occur.

- If either field is not a valid packed decimal number, a data exception (SOC 7) will occur.

- Sets the condition code

Code	Meaning
0	Result is 0
1	Result is negative
2	Result is positive
3	Overflow

Zero and Add Packed

Format: label ZAP $D_1(L_1,B_1),D_2(L_2,B_2)$

- The L_1 byte packed decimal field at $D_1(B_1)$ is zeroed out and the L_2 byte packed decimal number at $D_2(B_2)$ is added to the L_1 byte packed decimal number at $D_1(B_1)$. The sum is stored at $D_1(B_1)$.
- If the second operand is not a valid packed decimal number, a data exception (SOC 7) will occur.

- Sets the condition code

Code	Meaning
0	Result is 0
1	Result is negative
2	Result is positive
3	Overflow

- Use this instruction to move packed numbers: ZAP FLD1(3),FLD2(2)

- Use this instruction to initialize a field: ZAP FLD1(4),=P‘0’

Multiply Packed

Format: label MP $D_1(L_1,B_1),D_2(L_2,B_2)$

- The L_2 byte packed decimal number at $D_2(B_2)$ and the L_1 byte packed decimal number at $D_1(B_1)$ are multiplied together. The product is stored at $D_1(B_1)$.

- A specification exception (SOC 6) will occur if $L_2 > 8$ or if $L_2 >= L_1$.

- A data exception (SOC 7) will occur if the first L_2 bytes of the first operand are not all zeroes or if either field is not a valid packed decimal number.

Divide Packed

Format: label DP $D_1(L_1,B_1),D_2(L_2,B_2)$

- The L_2 byte packed decimal number at $D_2(B_2)$ is divided into the L_1 byte packed decimal number at $D_1(B_1)$. The quotient AND remainder are stored at $D_1(B_1)$.

- The length of the quotient is equal to (L_1 - L_2) bytes. The quotient is stored at $D_1(B_1)$ for (L_1 - L_2) bytes.

- The length of the remainder is L_2 bytes. The remainder is stored at $D_3(B_1)$ where D_3 = D_1 + (L_1 - L_2).

- A specification exception (SOC 6) will occur if L_2 > 8 or if L_2 >= L_1.

- A data exception (SOC 7) will occur if either field is not a valid packed decimal number.

- A decimal divide exception (S0C B) will occur if the quotient is too large or if the second operand is zero

Integer Division vs. floating Point Division

Using the divide packed (DP) instruction as it is results in integer division. For example, when you divide a 10-byte field, the dividend, by a 3-byte, the divisor, the quotient of the integer division is in the left 7 bytes of the 10-byte field and the remainder is in the rightmost 3.

There IS a way to "fake" real number, or floating point, division, though. If you shift the number in the the dividend field using the SRP instruction ***the number of decimal places to which you want to round the answer plus 1*** to the left before the division, you will get a real number result. Then, to round it, you shift just the quotient part of the answer one back to the right with rounding using SRP. You then use the edit pattern to place the decimal place where you want it in the output answer.

See the extended example at the end of this document.

Compare Packed

Format: label CP $D_1(L_1,B_1),D_2(L_2,B_2)$

- A numeric comparison of the L_2 byte packed decimal number at $D_2(B_2)$ and the L_1 byte packed decimal number at $D_1(B_1)$ is performed and the condition code is set.

Code	Meaning
0	Equality
1	Operand 1 is less than Operand 2
2	Operand 1 is greater than Operand 2

Shift and Round Packed

Format: label SRP $D_1(L,B_1),D_2(B_2),i$

- The L byte field at $D_1(B_1)$ is shifted.

- The amount and direction of the shift is determined by $D_2(B_2)$.

- i is used as the rounding factor. It is usually 0 or 5, but can be between 0 and 9. On a RIGHT shift, i is added to the leftmost digit shifted off the right. If the sum is greater than 9, the result is rounded up by 1.

- A shift to the LEFT is equivalent to multiplying by a power of 10

Left Shift Format: SRP $D_1(L,B_1),N,i$

N is a decimal number from 1 to 31 which is the number of positions to shift

If non-zero digits are lost on the shift, decimal overflow occurs

```
SRP   NUM(6),3,0    left shift by 3, which is equivalent to
                    multiplying by 10^3

Before execution:  NUM   00 00 01 20 75 9C
After execution:   NUM   00 12 07 59 00 0C
```

- A shift to the right is equivalent to dividing by a power of 10

Right Shift Format: SRP $D_1(L,B_1),(64-N),i$

N is a decimal number from 1 to 32 which is the number of positions to shift

```
SRP   NUM(6),(64-2),0    right shift by 2, which is
                         equivalent to dividing by 10^2

Before execution:  NUM   00 00 01 20 75 9C

After execution:   NUM   00 00 00 01 20 7C

SRP   NUM(6),(64-2),5

Before execution:  NUM   00 00 01 20 75 9C
```

After execution: NUM 00 00 00 01 20 8C

- Sets the condition code

Code	Meaning
0	Result is 0
1	Result is negative
2	Result is positive
3	Overflow

Formatting Packed Decimal Numbers for Printing

EDit Instruction

Format: label ED $D_1(L,B_1),D_2(B_2)$

- $D_1(B_1)$ is the address of an L byte field that initially contains a hexadecimal **pattern**. After execution of the instruction, this field will contain the formatted result.

- $D_2(B_2)$ is the **source field** and is the address of one or more contiguous packed decimal numbers to be formatted

- The pattern is made up of four types of characters

  1. X'20' **digit selector**
     Used to print one packed decimal digit

  2. X'21' **significance starter**
     Used to print one packed decimal digit and turns the significance indicator on **after** this byte

  3. X'22' **field separator**
     Used in formatting multiple packed numbers in one ED instruction

  4. Anything else is a **message character**. Common punctuation marks are found on page 35 of the yellow card.

- The **significance indicator** is used to indicate when leading zeroes should start to be printed. Initially is off.

- The first byte of the pattern is called a **fill character**. Leading zeroes or message characters that are to be suppressed are replaced by this character.

- The pattern and packed decimal number are both processed from left to right. The pattern is processed one BYTE at a time, while the packed decimal number is processed one DIGIT at a time.

- Execution proceeds as follows:

    - If the character from the pattern is a *digit selector*, a packed digit is examined.

        - If the significance indicator is off and the packed digit is a zero, the character in the pattern is replaced by the fill character

        - If the significance indicator is off and the packed digit is non-zero, the digit is converted to zoned format and the result replaces the character in the pattern. The significance indicator is turned on.

        - If the significance indicator is on, the packed digit is converted to zoned format and the result replaces the character in the pattern.

    - If the character from the pattern is a *significance starter*, the result is the same as above except that the significance indicator is always turned on AFTER the character in the pattern is replaced

    - If the character from the pattern is a *field separator*, it is replaced by the fill character and the significance indicator is turned off.

    - If the character from the pattern is a *message character*:

        - If the significance indicator is off, the character is replaced by the fill character

        - If the significance indicator is on, the message character is left unchanged

- A data exception (SOC 7) will occur if the second operand is not a valid packed decimal number

- Sets the condition code

Code	Meaning
0	The inspected character in the last field is 0

```
          1      The inspected character in the last field < 0
          2      The inspected character in the last field > 0
```

- Most ED instructions are preceded by a MVC that moves the pattern to $D_1(B_1)$

```
ED example 1 - Zero Suppression

  Source Field:   NUM    00 12 3C

  Pattern Field:  40 20 20 20 20 20   <= a space as fill character

  MVC   NUMOUT(6),=X'402020202020'
  ED    NUMOUT(6),NUM

  Output:    NUMOUT      40 40 40 F1 F2 F3

  When displayed:  ___123   where ___ is 3 spaces

ED example 2 - Zero Suppression

  Source Field:   NUM    00 00 0C

  Pattern Field:  40 20 20 20 20 20     <= a space as fill character

  MVC   NUMOUT(6),=X'402020202020'
  ED    NUMOUT(6),NUM
  Output:    NUMOUT    40 40 40 40 40 40

  When displayed:  6 spaces

ED example 3 - Zero Suppression with significance indicator

  Source Field:   NUM    00 00 0C

  Pattern Field:  40 20 20 20 21 20     <= a space as fill character

  MVC    NUMOUT(6),=X'402020202120'
  ED     NUMOUT(6),NUM

  Output:   NUMOUT    40 40 40 40 40 F0

  When displayed:    _____0    where _____ is 5 spaces

ED example 4 - Commas
```

```
  Source Field:   NUM     00 32 90 7C

  Pattern Field:  40 20 6B 20 20 20 6B 20 21 20

  MVC   NUMOUT(10),=X'40206B2020206B202120'
  ED    NUMOUT(10),NUM

  Output:    NUMOUT    40 40 40 40 F3 F2 6B F9 F0 F7

  When displayed:   ____32,907      where ____ is 4 spaces

ED example 5 - Decimal Point

  Source Field:   NUM     15 32 90 7C

  Pattern Field:  40 20 20 6B 20 21 20 4B 20 20

  MVC    NUMOUT(10),=X'4020206B2021204B2020'
  ED     NUMOUT(10),NUM

  Output:   NUMOUT     40 F1 F5 6B F3 F2 F9 4B F0 F7

  When displayed:    _15,329.07    where _ is a space

ED example 6 - Printing after a number

  Source Field:   NUM     90 7B
  Pattern Field:  40 20 21 20 60

  MVC    NUMOUT(5),=X'4020212060'
  ED     NUMOUT(5),NUM

  Output:   NUMOUT    40 F9 F0 F7 60

  When displayed:    _907-    where _ is a space

ED example 7 - Printing after a number

  Source Field:   NUM     90 7F

  Pattern Field:  40 20 21 20 60

  MVC    NUMOUT(5),=X'4020212060'
  ED     NUMOUT(5),NUM

  Output:   NUMOUT    40 F9 F0 F7 40
```

```
  When displayed:    _907_    where _ are spaces
```

ED example 8 - Printing more than one number

```
  Source Field:   NUM     36 0F 46 5F

  Pattern Field:  40 20 21 20 22 20 21 20

  MVC    NUMOUT(8),=X'4020212022202120'
  ED     NUMOUT(8),NUM

  Output:   NUMOUT    40 F3 F6 F0 40 F4 F6 F5

  When displayed:    _360_465    where _ are spaces
```

Edit and Mark Instruction

Format: label EDMK $D_1(L,B_1),D_2(B_2)$

- Performs exactly like the ED instruction but also sets a pointer to the first non-zero digit of an edited number.

- The address of the first non-zero digit is stored in the last 3 bytes of register 1 **ONLY** if a X'20' or X'21' was replaced by a source digit before a X'21' is reached

  Example 1: Floating dollar sign

  Source Field: NUM   46 78 23 9C

  Pattern Field: 40 20 20 6B 20 21 20 4B 20 20

```
  MVC   NUMOUT(10),=X'4020206B2021204B2020'
  EDMK  NUMOUT(10),NUM
  BCTR  R1,0
  MVI   0(R1),C'$'
```

  Output: NUMOUT   5B F4 F6 6B F7 F8 F2 4B F3 F9

  When displayed: $46,782.39

Since register 1 may not be altered by the EDMK instruction, it is a good idea to point register 1 to where the first non-blank character will occur.

Example 2:

```
   Source Field:   NUM    00 00 12 0C

   Pattern Field:  40 20 20 6B 20 21 20 4B 20 20

   LA    R1,NUMOUT+6   address of 1st non-zero digit you want to print
   MVC   NUMOUT(10),=X'4020206B2021204B2020'
   EDMK  NUMOUT(10),NUM
   BCTR  R1,0
   MVI   0(R1),C'$'

   When displayed:  _____$1.20       where _____ is 5 spaces
```

Example 3:

```
   Source Field:   NUM    00 00 0C

   Pattern Field:  40 20 21 20 4B 20 20

   LA    R1,NUMOUT+3   address of 1st non-zero digit you want to print
   MVC   NUMOUT(7),=X'402021204B2020'
   EDMK  NUMOUT(7),NUM
   BCTR  R1,0
   MVI   0(R1),C'$'

   When displayed:  __$0.00       where __ is 2 spaces
```

Convert to Binary

Format: label CVB R,D(X,B)

- The packed decimal number at D(X,B) is converted to its binary representation and stored in R
- D(X,B) is the address of an 8 byte field on a doubleword boundary
- A specification exception (SOC 6) will occur if D(X,B) is not on a doubleword boundary
- A data exception (SOC 7) will occur if the number at D(X,B) is not a valid packed decimal number
- A fixed point divide exception (SOC 9) will occur if the number at D(X,B) is too large to be represented in 32 bits

- This is the replacement for XDECI

  Example 1:

  DWORD is a double word whose contents are 00 00 00 00 00 00 01 0F

  CVB R4,DWORD will place 0000000A into R4

  Example 2:

```
Before CVB:    XDECI R5,BUFFER    getting a 6 digit stock number

With CVB:      PACK  TEMP(8),BUFFER(6)
               CVB   R5,TEMP

               TEMP  DS   D      to ensure a doubleword boundary
```

Convert to Decimal

```
Format: label   CVD     R,D(X,B)
```

- The binary number in R is converted to an 8 byte packed decimal number and stored starting at D(X,B)

- D(X,B) must be on a doubleword boundary

- A specification exception (SOC 6) will occur if D(X,B) is not on a doubleword boundary

- This is the replacement for XDECO

  Example 1:

  R7 contains FFFFFFFF

```
CVD   R7,DWORD       where DWORD  DS  D
```

  will place 00 00 00 00 00 00 00 1D into DWORD

  Example 2:

```
Before CVD:     XDECO R5,NUMOUT    where NUMOUT  DS  CL12

With CVD:       CVD   R5,TEMP
                MVC   NUMOUT(8),=X'4020202020202120'
                ED    NUMOUT(8),TEMP+4
```

```
TEMP   DS  D       ensure a doubleword boundary
NUMOUT DS  CL8

TEMP+4   is used so that the first 8 digits are
         skipped because they will all be zeroes
```

Example Packed Multiplication

The following is an example of calculating interest:

```
                    PACKED                    ACTUAL VALUES
Account Balance:    03 12 34 44 8F            0312344.48
Interest Rate:      x     00 27 5F            x    .0275
                 08 58 94 73 20 0F             $8,589.47
                 ^
                 If this packed decimal result is shifted and rounded four
                 digits to the right, it yields the correct answer and can
                 then be EDMK'd with the correct editing and two decimal
                 places.
```

Remember that the account balance from the input record must be packed into a 5-byte packed field (if it is zoned decimal or EBCDIC). It can then be EDMK'd directly into the print line from there. Then the 5-byte packed account balance can be ZAP'd into an 8-byte packed decimal field to prepare it for multiplication.

After the multiplication, the 8-byte field can be SRP'd four digits to the right with rounding and the answer will be in the rightmost five bytes of the 8-bytes packed decimal field, rounded to two decimal places!

Extended Example Packed Division

Numbers stored as packed decimal numbers are always stored as integers and have NO decimal point either stored or implied. Therefore, the programmer has to keep track of the decimal points so that the right results can be obtained when doing math with packed decimal numbers. This is usually most important when packed decimal numbers or the results of math need to be displayed or printed.

When an integer is divided by another, integer division occurs. The result is NOT like what results when two numbers are divided with a calculator. An integer quotient and an integer remainder are the results. The same thing happens when you simply divide one packed decimal number by another.

For example:

```
         ZAP   FIELD3(4),FIELD1(3)      GET THE DIVIDEND INTO FIELD3
         DP    FIELD3(4),FIELD2(1)      DIVIDE 45 BY

*  NOTE:  AFTER THIS DIVIDE, THE QUOTIENT WILL BE IN THE FIRST 3 BYTES OF
*  FIELD3 AND THE REMAINDER - ALWAYS THE SAME SIZE AS THE DIVISOR - WILL BE IN
*  THE LAST BYTE OF FIELD3.  HERE IS WHAT THE 4 BYTES OF FIELD3 LOOK LIKE
*  AFTER THE DIVISION:  00 01 0F 5F    NOTE THAT THERE ARE 2 PACKED DECIMAL
*  NUMBERS STORED IN FIELD3. THE FIRST THREE ARE THE QUOTIENT OF 10 AND THE
*  LAST BYTE IS THE REMAINDER OF 5.
```

and in storage:

```
FIELD1   DC    PL3'45'     STORAGE:  00 04 5F
FIELD2   DC    PL1'4'      STORAGE:  4F
FIELD3   DC    PL4'0'      STORAGE:  00 00 00 0F
```

`FIELD 3` is necessary to hold the result of the division. (Most commonly, just add the lengths of the field to be divided and the divisor.)

But, there is a way to turn this integer division into floating point division and get an answer similar to what would result using a calculator.

For example:

```
         SRP   FIELD2,2,0        FIRST, EVEN UP THE NUMBER OF IMPLIED DECIMAL
*                                PLACES.  BECAUSE FIELD1 HAS TWO IMPLIED,
*                                FIELD2 SHOULD HAVE THE SAME BEFORE DOING ANY
*                                THING ELSE
*
         ZAP   FIELD3(11),FIELD1(8)   COPY FIELD1 INTO FIELD3 TO PREPARE FOR
*                                     DIVISION
*
         SRP   FIELD3(11),3,0    SHIFT THE NUMBER TO BE DIVIDED 3 DIGITS TO
*                                THE LEFT TO ADD SOME "FAKE" DECIMAL PLACES.
*                                THE RULE IS:  SHIFT THE NUMBER OF DIGITS TO
*                                ROUND TO PLUS 1 MORE.
*
         DP    FIELD3(11),FIELD2(3)   NOW DIVIDE
         SRP   FIELD3(8),64-1,5       SHIFT JUST THE QUOTIENT PART ONE TO THE
*                                     RIGHT WITH ROUNDING.  THE RESULT IS NOW
*                                     ROUNDED TO TWO DECIMAL PLACES! JUST
*                                     REMEMBER THAT THE ANSWER DESIRED IS IN
*                                     THE FIRST 8 BYTES, THE QUOTIENT PART OF
*                                     FIELD3.  IGNORE THE REMAINDER FIELD.
*
```

and in storage:

```
FIELD1   DC    PL8'345622.97'    STORAGE:  00 00 00 03 45 62 29 7F
FIELD2   DC    PL3'23'           STORAGE:  00 02 3F
FIELD3   DC    PL11'0'           ONCE AGAIN, THE LENGTH OF THIS FIELD IS
*                                EQUAL TO THE LENGTHS OF THE TWO FIELDS
*                                INVOLVED IN THE DIVISION, FIELD1 AND FIELD2
```

Pitfalls

Before doing any of this, know the data and know it well!

For example, if the largest dollar amount ever to be divided is $99,999,999.99, a number which fits into a minimum number of 6 packed bytes. If the maximum number ever used to divide this by is $99.99, a number which fits into a minimum number of 3 packed bytes.

Do not let this trip you up, though! The programming might be tempted to think that the result will always fit into 6 bytes, right? But this is not true. To be a better Assembler programmer, it is important – and critical – to think in extremes: What if the largest dollar number is divided by the smallest divisor dollar amount? In other words, divide $99,999,999.99 by $0.01? The result would be $9,999,999,999.00 which would require 7 bytes minimum! AND! What if the result had to have more than just two implied decimal places? It could require even more than 7!

So, what is the moral of this story? ALWAYS KNOW THE DATA AND ANTICIPATE! This is what is commonly referred to as "defensive programming." Always take the largest amount ever to be divided and divide it by the smallest amount it will ever be divided by. Then, and only then, consider how many bytes are necessary before doing the divide pack!

Example:

```
         ZAP   FIELD3(10),FIELD1(6)     PREPARE FOR DIVIDE
         SRP   FIELD3(10),3,0           SHIFT 3 TO ROUND RESULT TO 2
         DP    FIELD3(10),FIELD2(3)     DIVIDE
         SRP   FIELD3(7),64-1,5         SHIFT 1 DIGIT RIGHT WITH ROUNDING

*  THE RESULT IS IN THE FIRST 7 BYTES OF FIELD3 AND ROUNDED TO AN IMPLIED 2
*  DECIMAL PLACES.  IT IS READY TO BE EDMK'd INTO THE OUTPUT FIELD.

FIELD1   DC    PL6'783452.75'   STORAGE:  00 07 83 45 27 5F
FIELD2   DC    PL3'3.98'        STORAGE:  00 39 8F
FIELD3   DC    PL10'0'          FIELD INTO WHICH DIVISION WILL BE DONE
```

5.19 USING, DROP and Dummy Sections

USING statement

Format: USING label,R

R contains an address

The USING statement tells the assembler to associate the address in R with label

It also tells the assembler which register to use when converting implicit addresses to explicit addresses.

A USING statements "reach" extends 4096 bytes.

DROP statement

Format: DROP $R_1,R_2,...,R_n$

Ends the domain of a USING statement

The DROP informs the assembler that registers $R_1,R_2,...,R_n$ are no longer to be associated with label

or

that the specified register is not supposed to be used to convert implicit addresses to explicit addresses.

A USING statement is in effect until a DROP statement is encountered. Any routine coded below a DROP statement will not know which register to use as a base register unless it contains a USING statement of its own.

Dummy SECTions

A dummy section is used to specify a format that can be associated with a particular area in storage without producing any object code

A dummy section begins with:

Format: label DSECT

The end of a dummy section is signaled by the occurrence of a CSECT statement, another DSECT statement, or an END statement.

An example DSECT:

```
      TABELEM DSECT
      STCKNUM DS    F
      ARTIST  DS    CL24
      TITLE   DS    CL24
      INSTOCK DS    F
      PRICE   DS    F
```

The above DSECT specifies the format for a table element. The labels STCKNUM, ARTIST, etc can be used rather than displacements

Before a DSECT can be used, a USING statement must be coded.

```
      USING TABELEM,R3
```

Before DSECTs

```
          LM    R3,R5,0(R1)
*
* R3 -> address of table
* R4 -> address of NAV storage area
* R5 -> address of input buffer
*
          XREAD 0(,R5),80
DO1       BL    ENDDO1
          XDECI R6,0(,R5)
          ST    R6,0(,R3)
          MVC   4(24,R3),7(R5)
          MVC   28(24,R3),32(R5)
          XDECI R6,57(,R5)
          ST    R6,52(,R3)
          XDECI R6,61(,R5)
          ST    R6,56(,R3)
          LA    R3,60(,R3)
          XREAD 0(,R5),80
          B     DO1
ENDDO1    DS    0H
          ST    R3,0(,R4)
```

After DSECTs

```
          USING TABELEM,R3
          LM    R3,R5,0(R1)
*
* R3 -> address of table
```

```
* R4 -> address of NAV storage area
* R5 -> address of input buffer
*
         XREAD 0(,R5),80
DO1      BL    ENDDO1
         XDECI R6,0(,R5)
         ST    R6,STCKNUM
         MVC   ARTIST(24),7(R5)
         MVC   TITLE(24),32(R5)
         XDECI R6,57(,R5)
         ST    R6,INSTOCK
         XDECI R6,61(,R5)
         ST    R6,PRICE
         LA    R3,60(,R3)
         XREAD 0(,R5),80
         B     DO1
ENDDO1   DS    0H
         ST    R3,0(,R4)
         DROP  R3
```

A second DSECT similar to the one below can be added to make the program even more readable:

```
INPUT   DSECT
INSTKNM DS    CL6
        DS    C
INART   DS    CL24
        DS    C
INTITLE DS    CL24
        DS    C
INAMT   DS    CL3
        DS    C
INPRICE DS    CL4
        DS    CL15
```

With the second DSECT

```
         USING TABELEM,R3
         USING INPUT,R5
         LM    R3,R5,0(R1)
*
* R3 -> address of table
* R4 -> address of EOT storage area
* R5 -> address of input buffer
*
         XREAD INPUT,80
```

```
DO1      BL    ENDDO1
         XDECI R6,INSTKNM
         ST    R6,STCKNUM
         MVC   ARTIST(24),INART
         MVC   TITLE(24),INTITLE
         XDECI R6,INAMT
         ST    R6,INSTOCK
         XDECI R6,INPRICE
         ST    R6,PRICE
         LA    R3,60(,R3)
         XREAD INPUT,80
         B     DO1
ENDDO1   DS    0H
         ST    R3,0(,R4)
         DROP  R3,R5
```

5.20 The Location Counter Value

Referencing the location counter value

The value of the location counter is referenced in Assembler code by using the asterisk (*).

When used in an instruction, the value of the asterisk resolves to the address of that instruction itself.

```
Loc Cntr Value           Assembler Code

000100                           B     PASTLOAD
000104                           L     R1,0(,R2)
000108                   PASTLOAD M     R4,=F'9'

000100                           B     *+8
000104                           L     R1,0(,R2)
000108                           M     R4,=F'9'
```

***+8** is interpreted as the value of the location counter before generation of the instruction plus 8

PRO: avoid cluttering up a program with a bunch of labels

CON: must be updated if instructions are added/deleted

Altering the location counter value

The value of the location counter can be altered by using the ORG instruction.

```
Format 1:   ORG   address     comment
```

- sets the location counter value to the specified address

```
Format 2:   ORG   ,           comment
```

- sets the location counter value to the next available unused location in the program

Similar to COBOL's REDEFINES clause

```
Example 1:  Suppose that the routine BUILD has to start at address
            000450.  It can be positioned at the address using ORG

Loc Cntr Value           Assembler Code

                                 ORG   X'000450'
000450                   BUILD   DS    0H
000450                           STM   R0,R15,BSAVE
000454                           LM    R3,R5,0(R1)
000458                           ...
```

```
Example 2:  Suppose that an input record can have 2 formats.  The
            format of the record depends on the first character.
            ORG can be used to define storage so NAMED areas of
            storage can be referenced in the code, rather than
            using displacements.

Name:  1         Code N          Address:  1         Code A
       2  - 16   First Name                2  - 41   Address
       17 - 31   Last Name                 42 - 61   City
       32 - 80   Unused                    62 - 63   State
                                           64 - 80   Unused
```

ASSEMBLER code before ORG:

```
Loc Cntr Value     Assembler Code
000080             BUFFER   DS    CL80      Input buffer
0000D0             PLINE    DC    CL1'-'    Print line for records
0000D1                      DC    CL131' '
```

In the code:

```
         XREAD BUFFER,80               Read a record
         CLI   BUFFER,C'N'             Check for name record
         BNE   ADDR                    Goto next check if not N
         MVC   PLINE+10(15),BUFFER+1   Move first name to print line
         MVC   PLINE+30(15),BUFFER+16  Move last name to print line
         B     PRINT                   Print the line
*
ADDR     CLI   BUFFER,C'A'             Check for address record
         BNE   ERROR                   Goto error routine if not A
         MVC   PLINE+12(40),BUFFER+1   Move address to print line
         MVC   PLINE+60(20),BUFFER+41  Move the city to print line
         MVC   PLINE+85(2),BUFFER+61   Move the state to print line
         B     PRINT                   Print the line
*
```

```
ERROR    MVC   PLINE+1,=C'*** Error: invalid record ***'
PRINT    XPRNT PLINE,132               Print the line
```

ASSEMBLER code after ORG:

```
Loc Cntr Value     Assembler Code

000080             BUFFER   DS    CL80
                            ORG   BUFFER    change LCV to 000080
000080             CODE     DS    CL1
000081             RESTREC  DS    CL79
                            ORG   RESTREC   change LCV to 000081
000081             FIRST    DS    CL15
000090             LAST     DS    CL15
                            ORG   RESTREC   change LCV to 000081
000081             ADDRESS  DS    CL40
0000A9             CITY     DS    CL20
0000BD             STATE    DS    CL2
                            ORG   ,         change LCV to 0000D0
0000D0             PLINE    DC    CL1'-'
```

In the code:

```
         XREAD BUFFER,80              Read a record
         CLI   CODE,C'N'              Check for name record
         BNE   ADDR                   Goto next check if not N
         MVC   PLINE+10(15),FIRST     Move first name to print line
         MVC   PLINE+30(15),LAST      Move last name to print line
         B     PRINT                  Print the line
*
ADDR     CLI   CODE,C'A'              Check for address record
         BNE   ERROR                  Goto error routine if not A
         MVC   PLINE+12(40),ADDRESS   Move address to print line
         MVC   PLINE+60(20),CITY      Move the city to print line
         MVC   PLINE+85(2),STATE      Move the state to print line
         B     PRINT                  Print the line
*
ERROR    MVC   PLINE+1,=C'*** Error: invalid record ***'
PRINT    XPRNT PLINE,132              Print the line
```

5.21 Conditional No Operation

Conditional No Operation, or CNOP, is used to align any instruction or storage area on a specific boundary.

Format: CNOP byte,word

- **byte** is an absolute expression that specifies at which even numbered byte in a fullword or doubleword the location counter is set to.
    - fullword: the value of the expression must be 0 or 2.
    - doubleword: the value of the expression must be 0, 2, 4, or 6.
- **word** is an absolute expression that specifies whether **byte** is a fullword or a doubleword.
    - fullword: the value of the expression must be 4.
    - doubleword: the value of the expression must be 8.
- Valid CNOP values: (also on page 19 of your yellow card).

	byte,word	alignment
CNOP	0,4	beginning of a fullword
CNOP	2,4	middle of a fullword
CNOP	0,8	beginning of a doubleword
CNOP	2,8	second halfword of a doubleword
CNOP	4,8	middle (third halfword) of a doubleword
CNOP	6,8	fourth halfword of a doubleword

- The location counter value is not altered if it is already aligned correctly.
- Alignment is achieved by filling in the skipped bytes with no-operation instructions (BCR 0,0).

5.22 Hashing and Hash Tables

Up to this point, you have been building tables sequentially and performing a linear search when a record was to be processed. This process is alright for smaller tables, but it becomes a waste of time when a table is large. So, a new random search, known as **hashing**, will be introduced.

Hashing can be used to both build and search a table. The basic idea behind hashing is to take a key and convert it through some fixed process to a **hash key** in the range of 0 to n-1, where n is the maximum number of slots in the table. The hash key will then be used to either store or find an item in the table.

There is the possibility that two keys could resolve to the same hash key. This situation is known as a **COLLISION**. When this occurs, the item will be stored in the next available slot in the table, assuming that the table is not already full.

The process of finding the next available slot when a collision occurs is known as a **LINEAR PROBE**. The linear probe is implemented via a linear search from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the top of the table and continue from there.

If an empty slot is not found before reaching the point of collision, the table is full.

The fixed process to convert a key to a hash key is known as a **hash function**. This function will be used whenever access to the table is needed.

The most common method of determining a hash key is the **division method of hashing**. The formula that will be used is:

```
hash key = key % number of entries in the table
```

The above formula uses the remainder of the division of the key by the number of entries in the table as the hash key

For example, if we assume a table that can hold 8 items:

```
hash key = key % number of entries in the table

   4     =  36 % 8
```

```
        2     =  18 % 8

        0     =  72 % 8

        3     =  43 % 8

        6     =   6 % 8

        2     =  10 % 8
```

Assembler does not allow subscripting/indexing so a displacement into the table must be calculated using the hash key. The formula for calculating a displacement into the table is:

```
   displacement = hash key * length of a table entry
```

The displacement can then be used to calculate the table entry address.

```
   table entry address = displacement + beginning address of the table
```

Hash Routine Pseudocode

```
searchKey is the key field of the entry to insert or the key that is
  being searched for

Calculate the table entry address
Save the table entry address

Set the return code to -1
Do while (the return code is -1)
  If an empty table slot is found
    Set the return code to 4
    Return the address of this table slot|
  Else
    If table entry's key is equal to searchKey
      Set the return code to 0
      Return the address of this table slot
    Else
      Increment the table pointer to the next entry
      If the physical end of table is reached
        Wrap around to the beginning of the table
      Endif
      If the table pointer is equal to saved table entry address
```

```
        Set the return code to 8
      Endif
    Endif
  Endif
Enddo
```

The return codes that are set in the pseudocode have a different meaning depending on whether an item is being inserted into the table or if the table is being searched for an item.

Return code meanings when **INSERTING** into the table:

Return Code	Meaning
0	A duplicate key has been found. DO NOT insert.
4	An empty slot has been found. OK to insert.
8	Table is full. DO NOT insert.

Return code meanings when **SEARCHING** the table for a key:

Return Code	Meaning
0	Successful Search. Key found.
4	Unsuccessful Search. Empty slot found.
8	Unsuccessful Search. Entire table has been searched.

5.23 More Character Instructions

Insert Character

This instruction is similar to the L instruction in that takes an area of memory and places it in a register. The main difference is that a single byte of characters will be inserted in the register rather than a fullword.

```
     Format:  label     IC    R,D(X,B)
```

A single byte specified by D(X,B) will be copied into the RIGHTMOST byte of the register specified by R.

The three LEFTMOST bytes of R are unchanged.

The condition code is not altered, and the only errors that can occur are protection and addressing exceptions.

Suppose FLAG DC   C'F' and that register 5 has the value
00 34 00 56. Execution of:

```
   IC   R5,FLAG
```

will change the contents of register 5 to 00 34 00 C6

While execution of:

```
   IC   R5,=X'03'
```

will change the contents of register 5 to 00 34 00 03.

Insert Characters under Mask

This instruction is similar to the IC instruction. The main difference is that 0 to 4 bytes will be inserted in the register rather than just a single character.

```
     Format:  label     ICM   R,mask,D(B)
```

Zero to four bytes will be copied from the storage area starting at D(B) and placed into the register specified by R. The number of bytes that are moved is determined by the mask. The bytes are placed into the register depending on the mask. The bytes that are copied from D(B) are contiguous bytes starting from the left.

Suppose FIELD DC   X'FFAABBCC' and that register 7 has the value A0 92 36 70. Execution of:

```
   ICM  R7,B'1001',FIELD
```

will change the contents of register 7 to FF 92 36 AA

While execution of:

```
   ICM  R7,B'0001',FIELD
```

will change the contents of register 7 to A0 92 36 FF. This is the same function as the IC instruction.

While execution of:

```
   ICM  R7,B'1111',FIELD
```

will change the contents of register 7 to FF AA BB CC. This behaves like a L instruction, but FIELD does not have to be on a fullword boundary.

The condition code is set.

Code	Meaning
0	All bytes inserted are 00 or the mask was 0
1	The LEFTMOST bit of the LEFTMOST inserted byte was 1
2	The LEFTMOST bit of the LEFTMOST inserted byte was 0 (but not all bits of the inserted bytes were 0)

Store Character

This instruction is similar to the ST instruction in that takes a value in a register and places it in an area of memory. The main difference is that a single byte from the register will be placed in memory rather than a fullword.

```
Format:  label     STC   R,D(X,B)
```

The RIGHTMOST byte of the register specified by R will be stored starting at D(X,B).

Unlike the ST instruction, D(X,B) does not need to be on a fullword boundary.

The condition code is not altered, and the only errors that can occur are protection and addressing exceptions.

Suppose FLD1 DC   C'FLAG'. Execution of:

```
   LA   R5,64           now R5 contains 00 00 00 40
   STC  R5,FLD1
```

will change the contents of FLD1 to 40 D3 C1 C7

Store Characters under Mask

This instruction is similar to the STC instruction. The main difference is that 0 to 4 bytes from the register will be placed in memory.

```
     Format:  label     STCM   R,mask,D(B)
```

The bytes of the register specified by R designated by the mask will be stored started as contiguous bytes starting at D(B).

The condition code is not altered.

Suppose register 7 contains A0 92 36 7C and that
FLD1 DC   4X'FF'. Execution of:

```
   STCM  R7,B'1011',FLD1
```

will change the contents of FLD1 to A0 36 7C FF

While execution of:

```
   STCM  R7,B'0001',FLD1
```

will change the contents of FLD1 to 7C FF FF FF. This is the same function as the STC instruction.

While execution of:

```
   STCM  R7,B'1111',FLD1
```

will change the contents of FLD1 to A0 92 36 7C. This is the same function as the ST instruction except that FLD1 does not need to be on a fullword boundary.

Compare Logical under Mask

This instruction is similar to the C instruction. The main difference is that 0 to 4 bytes from the register will be compared against bytes in memory.

```
     Format:  label     CLM   R,mask,D(B)
```

A logical comparison (treated as unsigned binary numbers) will be made between the bytes of the register designated by the mask and the same number of contiguous bytes starting at D(B).

The condition code is set.

Code	Meaning
0	The selected bytes are equal or the mask was 0
1	The register bytes are low
2	The contiguous bytes at D(B) are low

Suppose register 4 contains 00 AC 2B 40 and that

```
FLD1  DC   X'9F013C2F'.
```

Execution of:

```
   CLM   R4,B'0000',FLD1
```

has a comparison length of 0 bytes. The condition code is set to 0 since the mask is 0.

While execution of:

```
   CLM   R4,B'1010',FLD1
```

has a comparison length of 2 bytes. The condition code is set to 1 since 002B from register 4 is less than 9F01 from FLD1.

While execution of:

```
   CLM   R4,B'0111',FLD1
```

has a comparison length of 3 bytes. The condition code is set to 2 since 9F013C from FLD1 is less than AC2B40 from register 4.

5.24 Shift Instructions

The result of performing an *n* position right logical shift on a binary number containing *m* digits is obtained by:

1. Removing the RIGHTMOST *n* digits from the original number
2. Shifting the remaining digits *n* positions to the right
3. Placing *n* zeros to the left of the resulting number

For example, performing a three position, right logical shift on the number 10110001 results in:

```
BEFORE:    1 0 1 1 0 0 0 1

AFTER:     0 0 0 1 0 1 1 0
```

A left shift is performed in a similar manner.

Performing a three position, left logical shift on the number 10110001 results in:

```
BEFORE:    1 0 1 1 0 0 0 1

AFTER:     1 0 0 0 1 0 0 0
```

Shift Left Logical

```
     Format:  label     SLL   R,D(B)
```

The content of the register specified by R are shifted to the left depending on the rightmost six bits of the calculated D(B) address.

The condition code is not altered.

The digits that are shifted off are lost.

Suppose that register 7 has the value 0F 0F 0F 0F. Execution of:

```
   SLL  R7,2

           0    F    0    F    0    F    0    F

BEFORE:   0000 1111 0000 1111 0000 1111 0000 1111

AFTER:    0011 1100 0011 1100 0011 1100 0011 1100
```

```
           3    C    3    C    3    C    3    C
```

will change the contents of register 7 to 3C 3C 3C 3C.

Suppose that register 7 has the value 0F 0F 0F 0F and that register 2 has the value 00 00 00 05. Execution of:

```
   SLL  R7,0(R2)
```

```
0(R2)  =  0 + 000005  =  000005
```

The rightmost 6 bits of 0(R2) are 000101. When converted to decimal this is 5.

```
           0    F    0    F    0    F    0    F

BEFORE:   0000 1111 0000 1111 0000 1111 0000 1111

AFTER:    1110 0001 1110 0001 1110 0001 1110 0000

           E    1    E    1    E    1    E    0
```

will change the contents of register 7 to E1 E1 E1 E0.

Shift Right Logical

```
     Format:  label     SRL   R,D(B)
```

The content of the register specified by R are shifted to the right depending on the rightmost six bits of the calculated D(B) address.

The condition code is not altered.

The digits that are shifted off are lost.

Suppose that register 7 has the value 0F 0F 0F 0F. Execution of:

```
   SRL  R7,6
```

```
           0    F    0    F    0    F    0    F

BEFORE:   0000 1111 0000 1111 0000 1111 0000 1111

AFTER:    0000 0000 0011 1100 0011 1100 0011 1100

           0    0    3    C    3    C    3    C
```

will change the contents of register 7 to 00 3C 3C 3C.

Suppose that register 7 has the value 0F 0F 0F 0F and that register 2 has the value 00 00 00 05. Execution of:

```
   SRL  R7,0(R2)
```

```
0(R2)  =  0 + 000005  =  000005
```

The rightmost 6 bits of 0(R2) are 000101. When converted to decimal this is 5.

```
           0    F    0    F    0    F    0    F

BEFORE:   0000 1111 0000 1111 0000 1111 0000 1111

AFTER:    0000 0000 0111 1000 0111 1000 0111 1000

           0    0    7    8    7    8    7    8
```

will change the contents of register 7 to 00 78 78 78.

Shift Left Double Logical

```
     Format:  label     SLDL   R,D(B)
```

The content of the even-odd register pair specified by R are shifted to the left depending on the rightmost six bits of the calculated D(B) address.

The condition code is not altered.

The digits that are shifted off are lost.

The number being shifted is treated as a 64 bit number.

Suppose that register 6 has the value 00 00 F0 F0 and register 7 has the value 0F 0F 0F 0F. Execution of:

```
   SLDL  R6,12
```

```
BEFORE:   0000 0000 0000 0000 1111 0000 1111 0000

          0000 1111 0000 1111 0000 1111 0000 1111
```

```
AFTER:    0000 1111 0000 1111 0000 0000 1111 0000

          1111 0000 1111 0000 1111 0000 0000 0000
```

will change the contents of register 6 to 0F 0F 00 F0 and register 7 to F0 F0 F0 00.

Shift Right Double Logical

```
     Format:  label     SRDL   R,D(B)
```

The content of the even-odd register pair specified by R are shifted to the right depending on the rightmost six bits of the calculated D(B) address.

The condition code is not altered.

The digits that are shifted off are lost.

The number being shifted is treated as a 64 bit number.

Suppose that register 6 has the value 00 00 F0 F0 and register 7 has the value 0F 0F 0F 0F. Execution of:

```
   SRDL  R6,6

BEFORE:   0000 0000 0000 0000 1111 0000 1111 0000

          0000 1111 0000 1111 0000 1111 0000 1111

AFTER:    0000 0000 0000 0000 0000 0011 1100 0011

          1100 0000 0011 1100 0011 1100 0011 1100
```

will change the contents of register 6 to 00 00 03 C3 and register 7 to C0 3C 3C 3C.

The logical shift instructions are mainly used when you want to get rid of bits on the beginning or end of a number. They are also used when you want to position the bits in a register.

Shift Left Arithmetic

```
     Format:  label     SLA   R,D(B)
```

The content of the register specified by R are shifted to the left depending on the rightmost six bits of the calculated D(B) address.

An arithmetic left shift is equivalent to multiplying by a power of 2. The power of 2 to multiply by is specified by the second operand.

The first (or sign) bit (bit 0) does not participate in the shift.

If the bit shifted out of position 1 does not match the sign bit, overflow will occur. Therefore, a valid arithmetic shift will never alter the sign bit.

The condition code is set:

Code	Meaning
0	Result is 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

Suppose that register 7 has the value 00 00 00 05. Execution of:

```
   SLA  R7,2

           0    0    0    0    0    0    0    5

BEFORE:   0000 0000 0000 0000 0000 0000 0000 0101

AFTER:    0000 0000 0000 0000 0000 0000 0001 0100

           0    0    0    0    0    0    1    4
```

will change the contents of register 7 to 00 00 00 14.

Shift Right Arithmetic

```
     Format:  label     SRA   R,D(B)
```

The content of the register specified by R are shifted to the right depending on the rightmost six bits of the calculated D(B) address.

An arithmetic right shift is equivalent to dividing by a power of 2. The power of 2 to divide by is specified by the second operand.

The value of bits that are filled in on the left are equal to the sign bit. If the number is positive, the leftmost bits will be filled with

zero. If the number is negative, the leftmost bits will be filled with ones.

The condition code is set:

Code	Meaning
0	Result is 0
1	Result is less than 0
2	Result is greater than 0

Suppose that register 7 has the value FF FF FF FF. Execution of:

```
   SRA  R7,2

            F    F    F    F    F    F    F    F

BEFORE:   1111 1111 1111 1111 1111 1111 1111 1111

AFTER:    1111 1111 1111 1111 1111 1111 1111 1111

            F    F    F    F    F    F    F    F
```

will change the contents of register 7 to FF FF FF FF.
Remember that this is integer division.

Suppose that register 7 has the value 00 00 00 0C.
Execution of:

```
   SRA  R7,2

            0    0    0    0    0    0    0    C

BEFORE:   0000 0000 0000 0000 0000 0000 0000 1100

AFTER:    0000 0000 0000 0000 0000 0000 0000 0011

            0    0    0    0    0    0    0    3
```

will change the contents of register 7 to 00 00 00 03.

Shift Left Double Arithmetic

```
      Format:  label     SLDA   R,D(B)
```

The content of the even-odd register pair specified by R are shifted to the left depending on the rightmost six bits of the calculated D(B) address.

An arithmetic left shift is equivalent to multiplying by a power of 2. The power of 2 to multiply by is specified by the second operand.

The number being shifted is treated as a 64 bit number.

The first (or sign) bit (bit 0) does not participate in the shift.

If the bit shifted out of position 1 does not match the sign bit, overflow will occur. Therefore, a valid arithmetic shift will never alter the sign bit.

The condition code is set:

Code	Meaning
0	Result is 0
1	Result is less than 0
2	Result is greater than 0
3	Overflow

Suppose that register 6 has the value 00 00 00 00 and register 7 has the value 00 00 00 0A. Execution of:

```
   SLDA   R6,3

BEFORE:   0000 0000 0000 0000 0000 0000 0000 0000

          0000 0000 0000 0000 0000 0000 0000 1010

AFTER:    0000 0000 0000 0000 0000 0000 0000 0000

          0000 0000 0000 0000 0000 0000 0101 0000
```

will change the contents of register 6 to 00 00 00 00 and register 7 to 00 00 00 50.

Shift Right Double Arithmetic

```
     Format:  label     SRDA   R,D(B)
```

The content of the even-odd register pair specified by R are shifted to the right depending on the rightmost six bits of the calculated D(B) address.

An arithmetic right shift is equivalent to dividing by a power of 2. The power of 2 to divide by is specified by the second operand.

The value of bits that are filled in on the left are equal to the sign bit. If the number is positive, the leftmost bits will be filled with zero. If the number is negative, the leftmost bits will be filled with ones.

The condition code is set:

Code	Meaning
0	Result is 0
1	Result is less than 0
2	Result is greater than 0

Suppose that register 6 has the value 00 00 00 00 and register 7 has the value 00 00 01 00. Execution of:

```
   SRDA  R6,2

BEFORE:   0000 0000 0000 0000 0000 0000 0000 0000

          0000 0000 0000 0000 0000 0001 0000 0000

AFTER:    0000 0000 0000 0000 0000 0000 0000 0000

          0000 0000 0000 0000 0000 0000 0100 0000
```

will change the contents of register 6 to 00 00 00 00 and register 7 to 00 00 00 40.

5.25 Logical Instructions

The following instructions are going to perform **logical operations** between two fullword operands, a byte in storage and an immediate byte, or between two fields in storage.

A logical operation works on a bitwise level. Starting on the left, a bit from each operand has the logical operation performed and results in a boolean (true/false) value. This process proceeds until all of the bits have had the logical operation performed.

The possible logical operations are AND, OR, and EXCLUSIVE OR. Since these instructions work on a bit-wise level, a value of 0 is considered false and a value of 1 is considered true.

AND operation

AND Truth Table

Oper 1 Bit	Oper 2 Bit	Result
0	0	0
0	1	0
1	0	0
1	1	1

When an AND operation is performed, the result is 1 ONLY if both bits are 1; otherwise the result is 0.

The AND operation is used to set bits to 0

AND a bit with 1, it stays the same. AND a bit with 0, it becomes 0

```
RX Format:  label   N   R,D(X,B)
```

The fullword in the register and the fullword at D(X,B) are ANDed together. The result of the operation is placed in the register.

```
RR Format:  label   NR  R1,R2
```

The fullword in the R_1 and the fullword in R_2 are ANDed together. The result of the operation is placed in R_1.

```
SI Format:  label   NI  D(B),byte
```

The byte located at D(B) and the immediate byte specified by byte are ANDed together. The result of the operation is placed at D(B).

SS Format: label NC $D_1(L,B_1),D_2(B_2)$

The L byte storage areas at $D_1(B_1)$ and $D_2(B_2)$ are ANDed together. The result of the operation is placed starting at $D_1(B_1)$.

For all formats, if every bit of the result is 0, the condition code is set to 0; otherwise the condition code is set to 1.

AND Examples:

Suppose that a storage area called BYTE contains X'C5'. In binary:

```
          BYTE   1 1 0 0 0 1 0 1

     bit number  0 1 2 3 4 5 6 7
```

Execution of the instruction: NI BYTE,B'11111010'

```
          BYTE   1 1 0 0 0 1 0 1
                 1 1 1 1 1 0 1 0

        RESULT   1 1 0 0 0 0 0 0
```

will change or "turn off" bits 5 and 7 in BYTE. BYTE now contains X'C0'.

Suppose that register 5 contains 00 00 0A 07. Then execution of the instruction:

```
      N    R5,=X'00000000'

     Register 5   0000 0000 0000 0000 0000 1010 0000 0111
     X'00000000'  0000 0000 0000 0000 0000 0000 0000 0000
                  0000 0000 0000 0000 0000 0000 0000 0000
```

will zero out register 5.

While execution of N R5,=X'FFFFFFFF'

```
     Register 5   0000 0000 0000 0000 0000 1010 0000 0111
     X'FFFFFFFF'  1111 1111 1111 1111 1111 1111 1111 1111
                  0000 0000 0000 0000 0000 1010 0000 0111
```

will do nothing.

OR operation

OR Truth Table

Oper 1 Bit	Oper 2 Bit	Result
0	0	0
0	1	1
1	0	1
1	1	1

When an OR operation is performed, the result is 0 ONLY if both bits are otherwise the result is 1.

The OR operation is used to set bits to 1

OR a bit with 0, it stays the same. OR a bit with 1, it becomes 1

RX Format: label O R,D(X,B)

The fullword in the register and the fullword at D(X,B) are ORed together. The result of the operation is placed in the register.

RR Format: label OR R_1,R_2

The fullword in the R_1 and the fullword in R_2 are ORed together. The result of the operation is placed in R_1.

SI Format: label OI D(B),byte

The byte located at D(B) and the immediate byte specified by byte are ORed together. The result of the operation is placed at D(B).

SS Format: label OC $D_1(L,B_1)$,$D_2(B_2)$

The L byte storage areas at $D_1(B_1)$ and $D_2(B_2)$ are ORed together. The result of the operation is placed starting at $D_1(B_1)$.

For all formats, if every bit of the result is 0, the condition code is set to 0; otherwise the condition code is set to 1.

OR Examples:

Suppose that a storage area called BYTE contains X'05'. In binary:

BYTE 0 0 0 0 0 1 0 1

Execution of the instruction: OI BYTE,X'F0'

```
          BYTE   0 0 0 0 0 1 0 1
                 1 1 1 1 0 0 0 0

        RESULT   1 1 1 1 0 1 0 1
```

will change or "turn on" bits 0 through 3 in BYTE. So BYTE now contains X'F5' (the character representation of 5).

Suppose that NUM contains X'00378C'. Then execution of the instructions:

```
      UNPK NUMOUT(5),NUM(3)   results in NUMOUT  F0 F0 F3 F7 C8
      OI   NUMOUT+4,X'F0'

     NUMOUT+4 = C8   1100 1000
     X'F0'           1111 0000
                     1111 1000
```

The contents of NUMOUT is now F0 F0 F3 F7 F8.

Execution of O R5,=X'FFFFFFFF'

```
     Register 5   0000 0000 0000 0000 0000 1010 0000 0111
     X'FFFFFFFF'  1111 1111 1111 1111 1111 1111 1111 1111
                  1111 1111 1111 1111 1111 1111 1111 1111
```

will change the contents of register 5 to all Fs.

While execution of O R5,=X'00000000'

```
     Register 5   0000 0000 0000 0000 0000 1010 0000 0111
     X'00000000'  0000 0000 0000 0000 0000 0000 0000 0000
                  0000 0000 0000 0000 0000 1010 0000 0111
```

will do nothing.

EXCLUSIVE OR (XOR) operation

XOR Truth Table

Oper 1 Bit	Oper 2 Bit	Result
0	0	0
0	1	1
1	0	1
1	1	0

When an EXCLUSIVE OR operation is performed, the result is 1 if both bits are different; the result is 0 if both bits are the same.

The XOR operation is used to set bits to the opposite value.

XOR a bit with 0, it stays the same. XOR a bit with 1, it becomes the opposite value

```
     RX Format:  label   X   R,D(X,B)
```

The fullword in the register and the fullword at D(X,B) are XORed together. The result of the operation is placed in the register.

```
     RR Format:  label   XR  R1,R2
```

The fullword in the R_1 and the fullword in R_2 are XORed together. The result of the operation is placed in R_1.

```
     SI Format:  label   XI  D(B),byte
```

The byte located at D(B) and the immediate byte specified by byte are XORed together. The result of the operation is placed at D(B).

```
     SS Format:  label   XC  D1(L,B1),D2(B2)
```

The L byte storage areas at $D_1(B_1)$ and $D_2(B_2)$ are XORed together. The result of the operation is placed starting at $D_1(B_1)$.

For all formats, if every bit of the result is 0, the condition code is set to 0; otherwise the condition code is set to 1.

XOR Examples:

Suppose that a storage area called FLD1 contains X'C2C3'. In binary:

```
          FLD1   1100 0010 1100 0011

Execution of the instruction:     XC   FLD1(2),FLD1

          FLD1   1100 0010 1100 0011
          FLD1   1100 0010 1100 0011

   FLD1 RESULT   0000 0000 0000 0000
```

will set the field to all zeros. So, XORing a field against itself will set the field to all zeros.

Suppose that R5 contains 00 00 0A 07 and R7 contains 00 34 00 67. Then
execution of the instruction:

```
      XR   R5,R7

     Register 5   0000 0000 0000 0000 0000 1010 0000 0111
     Register 7   0000 0000 0011 0100 0000 0000 0110 0111
                  0000 0000 0011 0100 0000 1010 0110 0000
```

Now R5 contains 00 34 0A 60 and R7 contains 00 34 00 67

Execution of:

```
      XR   R7,R5

     Register 7   0000 0000 0011 0100 0000 0000 0110 0111
     Register 5   0000 0000 0011 0100 0000 1010 0110 0000
                  0000 0000 0000 0000 0000 1010 0000 0111
```

Now R5 contains 00 34 0A 60 and R7 contains 00 00 0A 07

Execution of:

```
      XR   R5,R7

     Register 5   0000 0000 0011 0100 0000 1010 0110 0000
     Register 7   0000 0000 0000 0000 0000 1010 0000 0111
                  0000 0000 0011 0100 0000 0000 0110 0111
```

Now R5 contains 00 34 00 67 and R7 contains 00 00 0A 07

The three XR instructions swapped the contents of the two registers.

5.26 Test Under Mask Instruction

The test under mask instruction is used to test the values of selected bits in a byte of storage.

```
     Format:  label     TM    D(B),mask
```

The value of the bits in the mask determine which bits of the byte at D(B) are tested. Bits that correspond to a 1 in the mask are tested, while bits that correspond to a 0 in the mask are not tested.

The condition code is set:

```
     Code    Meaning
      0      Every bit tested was 0
      1      Bits tested are a mix of 0s and 1s
      2      --
      3      Every bit tested was 1
```

Suppose that BYTE contains X'0E'. Execution of:

```
     TM    BYTE,B'10010000'

BYTE   X'0E' ->  B'00001110'
mask         ->  B'10010000'
```

will set the condition code to 0, since bits 0 and 3 of BYTE are both 0.

Execution of:

```
     TM    BYTE,B'10001000'

BYTE   X'0E' ->  B'00001110'
mask         ->  B'10001000'
```

will set the condition code to 1, since bits 0 is 0 and bit 4 is 1.

Execution of:

```
     TM    BYTE,B'00001110'

BYTE   X'0E' ->  B'00001110'
mask         ->  B'00001110'
```

will set the condition code to 3, since bits 4, 5, and 6 are all 1.

TM instructions are commonly followed by a branch instruction. The most common ones are:

```
     BZ    branch if all bits tested are ZERO

     BO    branch if all bits tested are ONE

     BM    branch if tested bits are MIXED

     BNZ   branch if tested bits are NOT ZERO

     BNO   branch if tested bits are NOT ONE

     BNM   branch if tested bits are NOT MIXED
```

5.27 MVCL, CLCL & EX Instructions

Move Character Long

This instruction behaves like MVC but it can be used to move fields as long as 2^{24}-1 bytes long.

```
Format:  label     MVCL  R1,R2
```

Each of the specified registers must be an even register that is used to designate an even-odd pair of registers. The contents of the field specified by R_2 are moved to the field specified by R_1.

R_1 EVEN register - the last 3 bytes specify the address of the destination field

R_1 ODD register - the last 3 bytes specify the length of the destination field

R_2 EVEN register - the last 3 bytes specify the address of the source field

R_2 ODD register - the last 3 bytes specify the length of the source field. The first byte is used as a pad character if the source field is shorter than the destination field.

If a byte is simultaneously in both the sending and receiving fields and if execution of the instruction causes the byte to receive a new value before the original contents are moved, then ***destructive overlap*** exists.

If destructive overlap occurs, no data is moved and the condition code is set.

Code	Meaning
0	The source and destination field are the same length
1	The source field is larger
2	The destination field is larger
3	Destructive overlap

After execution of the instruction, the values in each of the 4 registers will be changed to:

R_1 EVEN register is set to the address of the first byte past the end of the destination field

R_1 ODD register is set to 0

R_2 EVEN register is set to the address of the first byte beyond the last byte moved from the source field

R_2 ODD register is decremented by the number of bytes moved from the source field.

Suppose that R2 contains the address of a field and that R3 contains the length of that field. To set each byte of the field to X'00', the following code can be used:

```
LR    R4,R2      set the destination field address
SR    R5,R5      set the pad character to X'00' and length to 0
MVCL  R2,R4      puts the pad character of X'00' in the field
```

Compare Character Long

This instruction behaves like CLC but it can be used to compare fields as long as 2^{24}-1 bytes long.

Format: label CLCL R_1,R_2

Each of the specified registers must be an even register that is used to designate an even-odd pair of registers. The contents of the field specified by R_2 are compared against the field specified by R_1.

R_1 EVEN register - the last 3 bytes specify the address of the destination field

R_1 ODD register - the last 3 bytes specify the length of the destination field

R_2 EVEN register - the last 3 bytes specify the address of the source field

R_2 ODD register - the last 3 bytes specify the length of the source field. The first byte is used as a pad character. If the fields are of unequal length, the pad character is used to extend the shorter field for the comparison.

The condition code is set:

Code	Meaning
0	Fields are equal
1	Field specified by R_1 is low

2 Field specified by R_2 is low

The comparison occurs from left to right, one byte at a time

When two bytes are EQUAL, the even registers are incremented by 1 and the odd registers are decremented by 1. The only exception to this rule occurs when the one of the fields has been exhausted and the comparison is occuring with the pad character. In this case, only the register pair associated with the non-exhausted field are altered.

Execute

This instruction is used to execute other operations when the length of a field cannot be determined until execution time.

```
Format:  label     EX    R,D(X,B)
```

A copy of the instruction located at D(X,B) is made. The second byte of that copy is replaced by the rightmost byte of R. The copy of the instruction is then executed.

Neither the value in R nor the instruction at D(X,B) is changed.

If R is register 0, then the second byte is of the instruction is NOT changed, but it is still executed.

The byte to be replaced should initially contain X'00' if both hex digits are to be altered or X'0' in the place of the digit to be replaced.

Suppose that register 3 contains the address of a field that contains a zoned decimal number and that register 7 contains the length of the field. To pack the number into the field PACKNUM:

```
        BCTR     R7,0       decrement the length by 1
        EX       R7,PACKIT  pack the number
```

In storage:

```
PACKNUM DS      PL8
PACKIT  PACK    PACKNUM,0(0,R3)
```

The 0 within the parenthesis will be replaced at runtime by the rightmost hex digit of register 7.

Suppose that we want to add two packed decimal fields. The address of one field is in R3, while the length of that field is in R8. The address of the other field is in R4, while its length is in R9.

We want to add the value pointed to by R4 to the value pointed to by R3.

We need to get both lengths into the rightmost byte of one register so that both lengths can be replaced at runtime.

```
R8 = 00 00 00 07            R9 = 00 00 00 04

     BCTR R8,0           R8 now contains 00 00 00 06
     SLL  R8,4           R8 now contains 00 00 00 60
*
     BCTR R9,0           R9 now contains 00 00 00 03
*
     AR   R8,R9          R8 now contains 00 00 00 63
*
     EX   R8,ADD         will produce  FA63 3000 7000
*
*                        when decoded: AP  0(7,R3),0(4,R7)
```

In storage:

```
ADD   AP  0(0,R3),0(0,R7)   initially is FA00 3000 7000
```

Suppose that register 6 contains the address of a byte of storage that contains an unknown 8 bit number and that we want to test to see which bits are "on". A loop with an EXecuted TM instruction will be used. If a bit is "on", print the word "ON". If the bit is "off", print the word "OFF".

```
         LA    R3,8        there are 8 bits to test
         LA    R4,X'80'    start with bit 0
EXLOOP   EX    R4,TEST     test the bit
         BZ    BITOFF      if bit is 0, branch to print "OFF"
         XPRNT =C' ON',3   print "ON"
         B     NEXTBIT
BITOFF   XPRNT =C' OFF',4  print "OFF"
NEXTBIT  SRL   R4,1        Switch to next bit to test
         BCT   R3,EXLOOP   Repeat for all 8 bits
```

In storage:

```
TEST     TM    0(R6),0      Object code =  9100  6000
```

Loop Pass	EX instruction produces
1	91**80** 6000
2	91**40** 6000
3	91**20** 6000
4	91**10** 6000
5	91**08** 6000
6	91**04** 6000
7	91**02** 6000
8	91**01** 6000

5.28 TR and TRT Instructions

Translate

It may be necessary to replace each value in a field with a corresponding value that is determined by a fixed correspondence.

For example, suppose we want to change all of the lowercase letters in a character string to uppercase. Then:

```
   Original: ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Replacement: ABCDEFGHIJKLMNOPQRSTUVWXYZ ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In order to accomplish the conversion in ASSEMBLER, a 256 byte table must be constructed. Each byte that should be replaced in the string has its new value put into the table.

```
TRANTAB  DC    X'000102030405060708090A0B0C0D0E0F'
         DC    X'101112131415161718191A1B1C1D1E1F'
         ...
         ...
         DC    X'80C1C2C3C4C5C6C7C8C98A8B8C8D8E8F'
         DC    X'90D1D2D3D4D5D6D7D8D99A9B9C9D9E9F'
         DC    X'A0A1E2E3E4E5E6E7E8E9AAABACADAEAF'
         DC    X'B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF'
         ...
         ...
         DC    X'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF'
```

If it's known that the string to be converted has only lowercase letters, then the table can be rewritten using the ORG statement:

```
TRANTAB2 DC    256X'00'
         ORG   TRANTAB2+C'a'
         DC    X'C1C2C3C4C5C6C7C8C9'
         ORG   TRANTAB2+C'j'
         DC    X'D1D2D3D4D5D6D7D8D9'
         ORG   TRANTAB2+C's'
         DC    X'E2E3E4E5E6E7E8E9'
         ORG
```

The instruction to implement this process in ASSEMBLER is:

Format: label TR $D_1(L,B_1),D_2(B_2)$

The L byte field specified by $D_1(B_1)$ is translated using the translate table specified by $D_2(B_2)$.

```
     TR    STRING(5),TRANTAB2    changes hello to HELLO
*                        changes 88 85 93 93 96 to C8 C5 D3 D3 D6
In storage:

STRING   DC    C'hello'       in hex => 8885939396
TRANTAB2 DC    256X'00'
         ORG   TRANTAB2+C'a'
         DC    X'C1C2C3C4C5C6C7C8C9'
         ORG   TRANTAB2+C'j'
         DC    X'D1D2D3D4D5D6D7D8D9'
         ORG   TRANTAB2+C's'
         DC    X'E2E3E4E5E6E7E8E9'
         ORG
```

Translate and Test

It may be necessary to locate the first occurrence of a character in a field.

Format: label TRT $D_1(L,B_1),D_2(B_2)$

The L byte field specified by $D_1(B_1)$ is scanned for the character indicated in the translate table specified by $D_2(B_2)$.

The TRT instruction does NOT alter the characters in the original field.

Each byte of the designated field is scanned until a character is found that corresponds to a value in the translate table that is not 00.

If no character is found, the condition code is set to 0.

If a character is found, the following occurs:

1. The scanning terminates.
2. The last 3 bytes of register 1 is changed to the address of the detected character.
3. The rightmost byte of register 2 is changed to the value from the translate table.
4. If the character that is found is the last character of the field, the condition code is set to 2; otherwise, it is set to 1.

To find the first non-blank in the field specified by BUFFER,
we could

```
     XREAD BUFFER,80
     TRT   BUFFER(80),SCANTAB
     BZ    ALLBLANK
```

In storage:

```
BUFFER   DS    CL80
SCANTAB  DC    256X'FF'          All of the characters, except for
         ORG   SCANTAB+C' '        the space, are set to a non-zero
         DC    X'00'               value.
         ORG
```

Result:

If a non-blank is found, then:
   Register 1: last three bytes set to address of first non-blank
   Register 2: rightmost byte set to X'FF'
   Condition code is set to either 1 or 2
   Line after the branch instruction is executed

If no non-blank is found then:
   Register 1: unaltered
   Register 2: unaltered
   Condition code is set to 0
   Branch to ALLBLANK is taken

To find the first space in the field specified by BUFFER, we could

```
     XREAD BUFFER,80
     TRT   BUFFER(80),SPACETAB
     BZ    NOSPACES
```

In storage:

```
BUFFER   DS    CL80
SPACETAB DC    256X'00'         All characters, except for the space,
         ORG   SPACETAB+C' '       are set to zero.
         DC    X'FF'
         ORG
```

To find the first numeric digit or alphabetic character in the
field specified by BUFFER, we could

```
         SR    R2,R2
         XREAD BUFFER,80
         TRT   BUFFER(80),SCANTAB2
         B     BRTAB(R2)
BRTAB    B     NONE
         B     ALPHA
         B     DIGIT
```

In storage:

```
BUFFER   DS    CL80
SCANTAB2 DC    256X'00'
         ORG   SCANTAB2+C'A'      If alphabetic, set value in
         DC    9X'04'               register 2 to 04
         ORG   SCANTAB2+C'J'
         DC    9X'04'
         ORG   SCANTAB2+C'S'
         DC    8X'04'
         ORG   SCANTAB2+C'0'      If numeric, set value in
         DC    10X'08'              register 2 to 08
         ORG
```

Result:

If no alphabetic or numeric character is found, the value in register 2 remains 0 and the branch to NONE is taken.

If an alphabetic character is found, the rightmost byte of register 2 is set to 04 and the branch to ALPHA is taken.

If a numeric character is found, the rightmost byte of register 2 is set to 08 and the branch to DIGIT is taken.

5.29 Halfword Instructions

Up to this point, we have been working with instructions (ST, L, A, etc...) that work with 32-bit signed integers that are stored as fullwords. We're now going to look at a set of instructions that work with 16-bit signed integers that are stored as halfwords.

Halfword storage areas and constants are generated by using DS and DC statements with a storage class of H. Using the H will force halfword alignment.

Load Halfword

```
     Format: label    LH    R,D(X,B)
```

Copies into R the 32-bit representation of the number at the absolute address represented by D(X,B). The 2 leftmost bytes are set to X'0000' if the number is positive or X'FFFF' if the number is negative.

Execution of:

```
      LH    R4,=H'-12'
```

will change the contents of register 4 to X'FFFFFFF4'

Store Halfword

```
     Format: label    STH   R,D(X,B)
```

Places a copy of the rightmost 2 bytes of R at the absolute address represented by D(X,B).

Assuming that register 4 contains X'FFFFFFF4', execution of:

```
      STH    R4,HALF

HALF     DS   H
```

will change the contents of HALF to X'FFF4'

Compare Halfword

```
     Format: label    CH    R,D(X,B)
```

Compares the rightmost 2 bytes of R to the halfword at the absolute address represented by D(X,B) and sets the condition code.

```
      Code        Meaning
       0          Equality
       1          Halfword in R is low
       2          Halfword at D(X,B) is low
```

Assuming that register 4 contains X'FFFFFFF4', execution of:

```
      CH     R4,HALF

HALF     DC   H'5'
```

will set the condition code to 1 because the contents
of register 4 (-12) is less than the contents of HALF (5)

Add Halfword

```
      Format: label    AH    R,D(X,B)
```

The halfword at D(X,B) is added to the rightmost 2 bytes of R. The sum is stored in R. The condition code is set.

```
      Code        Meaning
       0          Sum is 0
       1          Sum is less than 0
       2          Sum is greater than 0
       3          Overflow - sum cannot be stored as a halfword
```

Assuming that register 4 contains X'FFFFFFF4', execution of:

```
      AH    R4,HALF

HALF     DC    H'13'
```

will change the contents of register 4 to X'00000001'

Subtract Halfword

```
      Format: label    SH    R,D(X,B)
```

The halfword at D(X,B) is subtracted from the rightmost 2 bytes of R. The difference is stored in R. The condition code is set.

```
      Code        Meaning
       0          Difference is 0
       1          Difference is less than 0
```

```
       2          Difference is greater than 0
       3          Overflow - difference cannot be stored as a halfword
```

Assuming that register 4 contains X'FFFFFFF4', execution of:

```
      SH    R4,HALF

HALF     DC    H'2'
```

will change the contents of register 4 to X'FFFFFFF2'

Multiply Halfword

Multiplying with halfwords is slightly different from multiplying with fullwords. A single register will be used rather than using an even-odd pair.

```
     Format: label    MH    R,D(X,B)
```

The halfword at D(X,B) and the rightmost 2 bytes of R are multiplied together. The result is stored as a 32-bit number in R.

Assuming that register 5 contains X'FFFFFFF4', execution of:

```
      MH    R5,HALF

HALF     DC    H'-2'
```

will change the contents of register 5 to X'00000018'

5.30 Macros

A macro is an extension to the basic ASSEMBLER language. They provide a means for generating a commonly used sequence of assembler instructions/statements. The sequence of instructions/statements will be coded ONE time within the macro definition. Whenever the sequence is needed within a program, the macro will be "called".

A macro definition precedes all CSECTs and DSECTs. It consists of four parts:

1. The macro instruction MACRO starting in column 10
2. The prototype statement (this line specifies the macro name and the arguments that it takes)
3. The macro body
4. The macro instruction MEND starting in column 10

MACRO signals the beginning of a macro definition.

Prototype format:

```
              label    macro-name     arguments (0 to 200 possible)
     Column:  1        10             16(usually)
```

The label in the prototype is optional. It can be used to put a label at the beginning of one of the lines of the loop body.

The macro body contains the macro instructions to be executed and the assembly instructions to be copied into the code.

MEND signals the end of a macro definition.

For example:

```
         MACRO
         EXMPL1
         LA    1,PARMLIST
         L     15,=V(BUILD)
         BALR  14,15
         MEND
```

To call the EXMPL1 macro:

```
         EXMPL1
```

In the assembly, you'll see:

```
+        LA    1,PARMLIST
+        L     15,=V(BUILD)
+        BALR  14,15
```

The + in column 1 indicates that the lines of code were in a macro.

Variable Symbols

Variable symbols are symbols that can be assigned values by either the programmer or the assembler. There are three types:

1. Symbolic Parameters
2. System Variables
3. SET Variables

Variable symbol names are:

1. Two to eight characters in length
2. The first character is ALWAYS an ampersand (&)
3. The second character is ALWAYS a letter
4. The rest of the characters are either letters or digits

Symbolic Parameters are used in a macro definition and are assigned a value by the programmer. When the macro is called, these parameters are replaced by the values that are assigned to them. There are two types of symbolic parameters:

1. Positional Parameters
2. Keyword Parameters

Positional parameters are symbolic parameters that must be specified in a specific order every time the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called.

```
         MACRO
&LABEL   EXMPL1  &SUBRTN,&PARMS
&LABEL   LA    1,&PARMS
         L     15,=V(&SUBRTN)
         BALR  14,15
         MEND
```

The new call of the EXMPL1 macro:

```
CALL1    EXMPL1 BUILD,PARM1
```

In the assembly:

```
+CALL1   LA    1,PARM1
+        L     15,=V(BUILD)
+        BALR  14,15
```

Keyword parameters are symbolic parameters that can be specified in any order when the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called. These parameters can be given a default value. If no default value is specified and if the parameter is not given a value when the macro is called, then the parameter will be replaced by a null string.

Each keyword parameter will have an equal sign (=) as the last character of the parameter name.

```
         MACRO
&LABEL   EXMPL1  &SUBRTN=BUILD,&PARMS=
&LABEL   LA    1,&PARMS
         L     15,=V(&SUBRTN)
         BALR  14,15
         MEND
```

The new call of the EXMPL1 macro:

```
         EXMPL1 PARMS=PARM2,SUBRTN=PRINT
```

In the assembly:

```
+        LA    1,PARM2
+        L     15,=V(PRINT)
+        BALR  14,15
```

```
         EXMPL1 PARMS=PARM3
```

In the assembly:

```
+        LA    1,PARM3
+        L     15,=V(BUILD)
+        BALR  14,15
```

```
         EXMPL1
```

In the assembly:

```
+        LA    1,
+        L     15,=V(BUILD)
+        BALR  14,15
```

If a combination of positional and keyword parameters is used, all of the positional parameters must be coded BEFORE the keyword parameters.

```
Prototype:  &LABEL  EXMPL2  &D,&E,&A=,&B=,&C=20

Call 1:     CALL1  EXMPL2  VAL3,VAL4,B=VAL1,A=(R5,R7),C=

   D will get VAL3, E will get VAL4, A will get (R5,R7),
   B will get VAL1 and C will get the null string

Call 2:     CALL2  EXMPL2  VAL3,,C=10,A=25

   D will get VAL3, E will get the null string, A will get 25,
   B will get the null string and C will get 10
```

System variables are variables that are assigned values by the assembler. They are defined by the system.

The most common system variables are: **&SYSNDX, &SYSLIST, &SYSDATE, &SYSTIME, &SYSPARM, &SYSECT**

&SYSNDX

- Generates a unique 4 digit number each time the macro is called
- The value is usually concatenated to a symbol of 4 characters
- The first value is 0001, with each subsequent call the value will be incremented by one

```
         MACRO
&LABEL   ADD   &NUM1,&NUM2
&LABEL   ST    5,SAVE
         L     5,&NUM1
         A     5,&NUM2
         ST    5,&NUM1
         B     NEXT
SAVE     DC    F'-1'
NEXT     L     5,SAVE
         MEND
```

In a program:

```
         ADD   FLD1,FLD2
+         ST    5,SAVE
+         L     5,FLD1
+         A     5,FLD2
+         ST    5,FLD1
+         B     NEXT
+SAVE     DC    F'-1'
+NEXT     L     5,SAVE
         ...
         ...
         ADD   FLD3,FLD4
+         ST    5,SAVE
+         L     5,FLD3
+         A     5,FLD4
+         ST    5,FLD3
+         B     NEXT
+SAVE     DC    F'-1'
+NEXT     L     5,SAVE
```

The labels SAVE and NEXT are duplicated when the macro is called for a second time, an assembler no-no. &SYSNDX can be used to solve this problem.

We're going to concatenate the &SYSNDX value onto the end of each of the labels.

```
         MACRO
&LABEL   ADD   &NUM1,&NUM2
&LABEL   ST    5,SAVE&SYSNDX
         L     5,&NUM1
         A     5,&NUM2
         ST    5,&NUM1
         B     NEXT&SYSNDX
SAVE&SYSNDX     DC    F'-1'
NEXT&SYSNDX     L     5,SAVE&SYSNDX
         MEND
```

In a program:

```
         ADD   FLD1,FLD2
+         ST    5,SAVE0001
+         L     5,FLD1
+         A     5,FLD2
+         ST    5,FLD1
+         B     NEXT0001
+SAVE0001     DC    F'-1'
```

```
+NEXT0001     L     5,SAVE0001
          ...
          ...
          ADD   FLD3,FLD4
+          ST    5,SAVE0002
+          L     5,FLD3
+          A     5,FLD4
+          ST    5,FLD3
+          B     NEXT0002
+SAVE0002     DC    F'-1'
+NEXT0002     L     5,SAVE0002
```

&SYSLIST

- Used to refer to a positional parameter or an entry in a positional parameter sublist

Format 1: &SYSLIST(n) -- refers to the n^{th} positional parameter

Format 2: &SYSLIST(n,k) -- refers to the n^{th} positional parameter's k^{th} member

```
CALL3    EXMPL3  ONE,TWO,(R5,R6,R7,R8),,FIVE,(R1)
```

(R5,R6,R7,R8) is a positional parameter sublist with 4 members

(R1) is a positional parameter sublist with 1 member

```
&SYSLIST(2)   -->  TWO
&SYSLIST(3,3) -->  R7
&SYSLIST(4)   -->  null value
&SYSLIST(6,1) -->  R1
```

Keyword parameters may also have sublists, but &SYSLIST CANNOT be used to reference the values. The name of the keyword parameter will be used.

```
          MACRO
          SUBLISTS  &P1,&KEY=(0,1,3)
&P1(1)    DC    F'&KEY(1)'
&SYSLIST(1,2)  EQU   &P1(2)
          MEND

&KEY    -->  (0,1,3)
&KEY(1) -->  0
&KEY(2) -->  1
```

```
&KEY(3) -->  3
&KEY(4) -->  NULL
```

In a program:

```
CALL4    SUBLISTS  (HERE,THERE),KEY=(5,6,7)
+HERE     DC    F'5'
+THERE    EQU   THERE
```

&SYSDATE is used to obtain the date that the source code was assembled. It is in the format MM/DD/YY

&SYSTIME is used to obtain the time that the source code was assembled. It is in the format hh.mm

&SYSECT is used to obtain the name of CSECT that the macro was called in.

&SYSPARM is used to pass a character string from the JCL.

SET variables are parameters that are assigned an initial value and can be altered during the macro call. There are three types:

1. Arithmetic
2. Binary
3. Character

Each SET variable can be either ***local*** or ***global***.

A local SET variable is assigned an initial value every time the macro is called. It is known only to the macro it is created in.

A global SET variable is initialized the first time the macro is called and retains a value from one call to another. They can be referenced within other macros also, but they must be declared within each macro.

All SET variables must be declared directly after the prototype statement. All global variables must be declared before local variables.

Arithmetic SET variables are variables whose value can be altered by an arithmetic expression.

To declare a LOCAL variable: `LCLA  &var_name1,&var_name2...`

- These are initialized to 0 every time the macro is called

To declare a GLOBAL variable: GBLA &var_name1,&var_name2...

- These are initialized to 0 the first time the macro is called

To alter the value: &var_name SETA arithmetic_expression

```
         LCLA  &CNTR     declares an arithmetic variable called &CNTR
&CNTR    SETA  1         changes the counter value to 1
&CNTR    SETA  &CNTR+1   increments the counter value by 1
&CNTR    SETA  &CNTR*5   multiply counter by 5
```

Binary SET variables are variables whose value can be either 0 or 1. They are typically used as flag variables with 0 being FALSE and 1 being TRUE.

To declare a LOCAL variable: LCLB &var_name1,&var_name2...

- These are initialized to 0 every time the macro is called

To declare a GLOBAL variable: GBLB &var_name1,&var_name2...

- These are initialized to 0 the first time the macro is called

To alter the value: &var_name SETB (condition)

- (condition) will resolve to either a 0 (false) or 1 (true) because of this, 0 or 1 can be hard-coded in place of the (condition)

```
         LCLB  &FLAG
&FLAG    SETB  ('&PARMS' EQ '')
```

Character SET variables are variables whose value can be a text string of up to 255 characters.

To declare a LOCAL variable: LCLC &var_name1,&var_name2...

- These are initialized to the null string every time the macro is called

To declare a GLOBAL variable: GBLC &var_name1,&var_name2...

- These are initialized to the null string the first time the macro is called

To alter the value: &var_name SETC string

- string can be:
  1. A text string enclosed in single quotes 'hello'
  2. A text string and character set symbol 'ABC&MSG' or '&MSG.ABC'
  3. A combination of two character set symbols '&MSG1&MSG2'
  4. A substring of a character set symbol '&MSG'(start,length)
     - start is the first character of substring
     - length is the length of the substring

```
         LCLC  &MSG
&MSG     SETC  'Hello'
&MSG     SETC  '&MSG. world'   changes &MSG to 'Hello world'
```

NOTE: The period is needed to signal the end of the set symbol name

```
         LCLC  &NDX
&NDX     SETC  '&SYSNDX'      now &NDX can be used in place of &SYSNDX
         LCLC  &TEXT1,&TEXT2
&TEXT1   SETC  'BOBCAT'
&TEXT2   SETC  '&TEXT1'(1,3)    &TEXT2 set to 'BOB'
```

There are two types of comments that can be included in a macro definition.

A comment starting with a * in column 1 will be reproduced in the assembly.

A comment starting with a .* will NOT be reproduced in the assembly.

```
         MACRO
&LABEL   EXMPL1  &SUBRTN=BUILD,&PARMS=
.*************************************************************
.*
.* Definition for a macro to call an external subroutine
.*
.*************************************************************
*
* The following lines will call an external subroutine
*
&LABEL   LA    1,&PARMS
         L     15,=V(&SUBRTN)
         BALR  14,15
         MEND
```

The new call of the EXMPL1 macro:

```
         EXMPL1 SUBRTN=PRINT,PARMS=PARM2
```

In the assembly:

```
+*
+* The following lines will call an external subroutine
+*
+        LA    1,PARM4
+        L     15,=V(PRINT)
+        BALR  14,15
```

A couple more macro instructions:

MEXIT

This instruction terminates the processing of a macro definition.

Any instructions/statements that follow an MEXIT will NOT be part of the assembly listing.

MEXIT is usually used when error checking is being done within the macro body.

```
         MACRO
&LABEL   EXMPL1  &SUBRTN=BUILD,&PARMS=
&LABEL   LA    1,&PARMS
         L     15,=V(&SUBRTN)
         MEXIT
         BALR  14,15
         MEND
```

The call of the EXMPL1 macro:

```
         EXMPL1 SUBRTN=PRINT,PARMS=PARM2
```

In the assembly:

```
+        LA    1,PARM4
+        L     15,=V(PRINT)
```

MNOTE

This instruction is used to generate an error message in an assembly listing.

```
Format 1:    MNOTE   'error message goes here'
```

If the MNOTE is executed, the error message will be displayed.

```
Format 2:    MNOTE   severity,'error message goes here'
```

This format of MNOTE assigns a severity code to the error. The severity code can be between 0 and 255 and defaults to 1 if it is not specified. The code is used to indicate if the message is an error or just a warning.

If this version of MNOTE is executed, the error message will be displayed along with the severity code.

Conditional Assembly

The sequence that the instructions/statements are processed in can be altered by using the conditional assembly instructions.

A **sequence symbol** is any symbol that satisifes the following rules:

- Two to eight characters in length
- The first character is a period
- The second character is a letter
- The remaining characters are either letters or digits

Sequence symbols are equivalent to a label in an assembler program.

ANOP

This instruction provides a label for other instructions to branch to. It is equivalent to a DS 0H in an assembler program.

```
     Format:  .sequence_symbol   ANOP
```

AGO

This instruction performs an unconditional branch.

```
     Format:  .sequence_symbol_1   AGO   .sequence_symbol_2
```

An unconditional branch to .sequence_symbol_2 is taken. .sequence_symbol_1 is optional. If .sequence_symbol_1 is specified, it provides a label for another conditional assembly instruction to branch to.

AIF

This instruction performs a conditional branch.

```
     Format:  .seq_sym_1   AIF   (conditional expression).seq_sym_2
```

A branch to .seq_sym_2 is taken if the result of the conditional expression is 1 (equivalent to TRUE).

If the conditional expression evaluates to 0 (equivalent to FALSE), the line immediately following the AIF is executed.

AIF conditional expression

```
     Format:  (operand_1  relational_operator  operand_2)
```

operand_1 and operand_2 may be:

- an arithmetic expression
- 0 or 1
- a character string enclosed in single quotes
- a variable symbol

relational_operator may be:

- EQ for equality
- NE for not equal
- LT for less than
- LE for less than or equal to
- GT for greater than
- GE for greater than or equal to

For example:

```
       AIF    ('&PARMS' EQ '').NOPARM
```

If &PARMS is equal to the null string, branch to .NOPARM

```
         MACRO
&LABEL   EXMPL1  &SUBRTN=BUILD,&PARMS=
         AIF   ('&PARMS' EQ '').NOPARMS
&LABEL   LA    1,&PARMS
.NOPARMS ANOP
         L     15,=V(&SUBRTN)
         BALR  14,15
         MEND
```

The call of the EXMPL1 macro:

```
         EXMPL1 SUBRTN=PRINT
```

In the assembly:

```
+        L     15,=V(PRINT)
+        BALR  14,15
```

```
         MACRO
         EQUREGS
         LCLA    &NUM
.LOOP    AIF     (&NUM  GT  15).LPEND
R&NUM    EQU     &NUM
&NUM     SETA    &NUM+1
         AGO     .LOOP
.LPEND   ANOP
         MEND
```

The call of the EQUREGS macro:

```
         EQUREGS
```

In the assembly:

```
+R0      EQU   0
+R1      EQU   1
...
...
+R15     EQU   15
```

```
         MACRO
         EXITLINK   &TYPE
         AIF   ('&TYPE'  NE  '').FOUND
         MNOTE '*** MISSING TYPE PARAMETER ***'
         MEXIT
.FOUND   ANOP
         AIF   ('&TYPE'  EQ  'N').NORMAL
         AIF   ('&TYPE'  EQ  'R').RETURN
         AIF   ('&TYPE'  EQ  'V').VALUE
         AIF   ('&TYPE'  EQ  'B').BOTH
         MNOTE 8,'*** TYPE IS NOT VALID'
         MEXIT
.NORMAL  ANOP           <--- NORMAL LINKAGE
```

```
         L     13,4(,13)
         LM    14,12,12(13)
         BR    14
         AGO   .DONE
.RETURN  ANOP            <--- RETURN CODE IN REGISTER 15
         L     13,4(,13)
         L     14,12(,13)
         LM    0,12,20(13)
         BR    14
         AGO   .DONE
.VALUE   ANOP            <--- CALCULATED VALUE IN REGISTER 0
         L     13,4(,13)
         LM    14,15,12(13)
         LM    1,12,24(13)
         BR    14
         AGO   .DONE
.BOTH    ANOP            <--- VALUES IN REGISTER 15 AND 0
         L     13,4(,13)
         L     14,12(,13)
         LM    1,12,24(13)
         BR    14
.DONE    ANOP
         MEND
```

The call of the EXITLINK macro:

```
         EXITLINK
```

In the assembly:

```
+ *** MISSING TYPE PARAMETER ***
```

Another call:

```
         EXITLINK  CHAR
+
+        L     13,4(,13)
+        L     14,12(,13)
+        LM    0,12,20(13)
+        BR    14

CHAR     DC    C'R'
```

Data Attributes

For each constant or instruction the assembler assigns **data attributes**. These attributes can be used with the conditional assembly instructions to control the sequence and contents of the statements generated.

Length Attribute

This attribute is a numeric value that is equal to the number of bytes occupied by the data that is represented by a symbolic parameter.

The value of the symbolic parameter MUST be a label in the calling program.

Format: L'symbolic_parameter

```
         MACRO
         MOVEIT   &DEST,&SOURCE
         MVC   &DEST.(L'&SOURCE),&SOURCE
         MEND
```

In a program:

```
         MOVEIT  PLINE,TABLE
+        MVC   PLINE(80),TABLE


         MOVEIT  PLINE,80
+        MVC   PLINE(0),80
```

The length attribute is 0 because 80 is not a label in the calling program.

```
         MOVEIT  PLINE,TABLE2
+        MVC   PLINE(20),TABLE2
```

The length attribute is 20. 4 is the repetition factor so it is NOT included in the length.

```
PLINE    DS    CL132
TABLE    DS    CL80
TABLE2   DS    4CL20
```

Count Attribute

This attribute is a numeric value that is equal to the number of characters in the actual parameter being passed.

If the value is 0, the parameter is missing.

Format: K'symbolic_parameter

```
         MACRO
         PRINTIT   &P1,&P2
         LCLA      &CNT1,&CNT2
         AIF       (K'&P1  NE  0).FOUND
         MNOTE     '*** MISSING PARAMETER ***'
         MEXIT
.FOUND   ANOP
&CNT1    SETA      K'&P1
&CNT2    SETA      K'&P2-2
         XPRNT =C&P2,&CNT2
         LA    5,&CNT1
         MEND
```

In a program:

```
         PRINTIT  JUNKY,'1  TOP OF PAGE'
+        XPRNT =C'1  TOP OF PAGE',14
+        LA    5,5
```

K'&P1 --> K'JUNKY --> 5 since there are 5 letters in JUNKY

K'&P2-2 --> K''1 TOP OF PAGE'-2 --> 16-2 --> 14
  the 2 is subtracted to account for the single quotes

```
         PRINTIT  WHATEVER,'0  DOUBLE SPACE'
+        XPRNT =C'0  DOUBLE SPACE',15
+        LA    5,8
```

Number Attribute

This attribute is a numeric value that is equal to the number of operands in an operand sublist.

If this attribute is used on &SYSLIST, will give the number of positional parameters on the prototype statement.

If this attribute is used on &SYSLIST(m), will give the number of items in the sublist of the m^{th} parameter.

If this attribute is used on any other operand, will give the number of items in the sublist of that parameter.

Format: N'symbolic_parameter

The macro that follows is going to generate a branch table similar to one used in the hashing assignment. &BTAB is a sublist of the different labels to branch to. The first member is for a return code of 0, the second a return code of 4, etc. If &BTAB is not specified when the macro is called, do not generate the table. If &BTAB is specified, you may assume that there is at least one member in the sublist. The table that is generated should have a unique label (BTAB????) in case the macro is called more than one time. After all of the necessary branches have been created, a SOC1???? DC   H'0' should be coded so that the program abends if an improper return code is returned in register 15.

```
         MACRO
         BRANCHTAB &BTAB
         LCLC    &NDX
         LCLA    &NUM,&CNT
&NDX     SETC    '&SYSNDX'
&CNT     SETA    2
&NUM     SETA    N'&BTAB
         AIF     ('&BTAB'  EQ  '').NOTAB
         B     BTAB&NDX.(15)
BTAB&NDX B     &BTAB(1)
.BLOOP   AIF     (&CNT  GT  &NUM).DONE
         B     &BTAB(&CNT)
&CNT     SETA    &CNT+1
         AGO     .BLOOP
.DONE    ANOP
SOC1&NDX DC    H'0'
.NOTAB   ANOP
         MEND
```

In a program:

```
         BRANCHTAB   (RC0,RC4,RC8)
+         B     BTAB0001(15)
+BTAB0001 B     RC0
+         B     RC4
+         B     RC8
```

```
+SOC10001 DC    H'0'
```

Type Attribute

This attribute indicates the type of data of the field assigned to the symbolic parameter when the macro is called.

Format: T'symbolic_parameter

Symbols for DC/DS statement	Meaning
A	Address constant
B	Binary constant
C	Character constant
F	Fullword
H	Halfword
P	Packed decimal
R	A-con or V-con
X	Hexadecimal
Z	Zoned decimal
V	V-con

Symbols for other statement	Meaning
I	Machine Instruction
J	CSECT name
M	Macro Instruction
O	Omitted Operand

```
         MACRO
         TEXMPL     &P1,&P2
         LCLC       &CH1,&CH2
         AIF  (T'&P1  NE  'O').FOUND
         MNOTE   '**** PARAMETER IS MISSING ****'
         MEXIT
.FOUND   ANOP
&CH1     SETC  T'&P2
&CH2     SETC  T'&P1
&CH2     SETC  '&CH2&CH1'
         DC    C'&CH2'
         MEND
```

In a program:

```
         TEXMPL   TABLE,PLINE
+         DC    C'FC'
TABLE    DS    20F
PLINE    DC    CL12
```

Appendix A - Assist's X-Type Instructions

XDUMP

Used to "dump" the GPRs or program storage.

Format 1: XDUMP comment

- produces a hexadecimal dump of the GPRs

Format 2: XDUMP D(X,B),length

- produces a hexadecimal dump of storage starting from D(X,B) for length bytes
- a label can be substituted for D(X,B)
- an expression that resolves to an integer value can be substituted for length

If the line **XDUMP NUM1,RESULT2+4-NUM1** was coded somewhere in the program, a hexadecimal dump of the entire storage area would be produced.

XREAD

Used to read an input record from the file indicated on FT05F001 DD card or from instream data.

Format: label XREAD D(X,B),length

- reads length bytes into the input buffer located at D(X,B)

Sets the Condition Code:

Code	Meaning
0	Successful read
1	End of file reached

Example: XREAD BUFFER,80

Assuming that BUFFER DS CL80 is located in storage.

XPRNT

Used to print a print line of output.

Format: label XPRNT D(X,B),length

- prints length bytes of the print line located at **D(X,B)**

- Print lines must be explicitly defined in program storage

- usually 132 characters long

- 1st character is used for carriage control:

space	single spacing
zero (0)	double spacing
hypen (-)	triple spacing
number 1	new page

Example:

```
         XPRNT LINE1,132   Print the line of output named LINE1

>>> more code here <<<

LINE1    DC    CL1' '
         DC    CL16'Num1 is equal to'
NUM1     DS    CL12
         DC    CL5' '
         DC    CL16'Num2 is equal to'
NUM2     DS    CL12
         DC    CL70' '
```

XDECI

Used to convert a number from its character representation to its binary representation so that math can be performed.

Format: label XDECI R,D(X,B)

- converts the number at D(X,B) to binary and stores it in R

- Stops scanning when a space is reached

- 1st character may be a digit or a plus (+) or minus (-) sign

- Register 1 is set to the address of the next value to read

Sets the Condition Code:

Code	Meaning
0	Converted number is 0
1	Converted number is less than 0
2	Converted number is greater than 0
3	Attempt to convert an invalid number

```
Example:  XDECI 3,BUFFER   Convert 1st number on an input record
          XDECI 4,0(,1)    Convert 2nd number on the input record
```

XDECO

Used to convert a number from its binary representation to its character representation so it can be printed,

Format: label XDECO R,D(X,B)

- converts the binary number in register **R** to printable characters and stores it at **D(X,B)**
- Number is converted to a 12 byte character representation
- If number is negative, a minus sign is printed to left of 1st digit

```
Examples:  XDECO 3,NUM1   Convert number in register 3 to printable
                          format and place it in NUM1, which is
                          defined as NUM1     DS    CL12

           XDECO 4,NUM2   Convert number in register 4 to printable
                          format and place it in NUM2, which is
                          defined as NUM2 DS CL12
```

Appendix B - ABENDS, Dumps and the PSW

If your program **ABEND**s (**AB**normally **END**s), ASSIST generally will provide you with a memory dump which can help you isolate the reason for the ABEND. The only instances in which you will not get a dump are those in which your job exceeds the time limit (no time remains to generate the dump), or the job generates more than the number of lines allowed (2000 is the default maximum number.)

The information available to you in a dump includes:

The contents of the PSW.

See Program Status Word (PSW) below

The completion code.

See your yellow card for a listing of program interruption codes - also, appendix D of your text gives a good explanation of the more commonly encountered completion codes, along with various programmer errors which can cause those types of interrupt.

A trace of the last few instructions executed.

A trace of the last few branch instructions executed.

Contents of the 16 general purpose registers

At the time of the ABEND. Since you are not going to be using the floating point registers, you can ignore them.

The contents of user storage

The portion of main storage used by your program and its save areas is dumped in hexadecimal. Each line contains 32 bytes of storage. In the left-hand margin you will find the address of the first of these 32 bytes (LOC.) On the right-hand margin you will find (between two *s) a translation into character form, where alphabetic and numeric characters and blanks are identified whenever a byte contains the character's encoded form a period is printed to represent any other byte values.

When your program ABENDs, you should be able to answer questions such as:

1. What was the reason for the ABEND (interruption code)?
2. What does that interruption code mean?
3. What was the last instruction executed?
4. Did the registers contain the "right" values?
5. Were the contents of user storage correct?

Program Status Word (PSW)

The Program Status Word or PSW is a collection of data 8 bytes (or 64 bits) long, maintained by the operating system. It keeps track of the current state of the system.

We can usually ignore the PSW unless an ABEND has occurred. When an ABEND does occur, ASSIST will print out various information for us including the PSW. The PSW is printed out as 16 hex digits in two groups of 8.

What information is in the PSW?

You can find a detailed list of the fields in the PSW in the yellow card. We use the "BC Mode" of the PSW. Here is a list of some fields we will need in this course:

Bytes	**Contents**
1 & 2:	assorted data we can ignore for now
3 & 4:	Interruption Code
5:	2 bits = Instruction Length Code (ILC)
	2 bits = Condition Code (CC)
	4 bits we can ignore for now
6 - 8:	Address of the next instruction

What are all these?

- The Interruption Code indicates the type of ABEND that has occurred.
- The ILC gives us the length of the current instruction, measured in halfwords,
- The CC gives us the condition code as set most recently.
- The address of the next instruction gives us the location of the instruction that would have been executed if the program had not ABENDed.

So what do we do with all this?

One problem with an ABEND is to determine which instruction caused the ABEND. We can find it using the information in the PSW:

Address of ABENDing instruction = Address of next instruction - 2 * ILC

Example

Suppose an ABEND occurs and the PSW has the value `FFC50001 8000001A`.

We can look at this and know that:

- The interrupt code is `0001` (Operation Exception).
- The ILC is `10` (binary) or 2 (decimal), so the ABENDing instruction is 4 bytes long.
- The CC is `00`.
- The address of the next instruction is `00001A`.
- The address of the ABENDing instruction is `00001A - 4 = 000016`.

Dump Example 1

Type in and run the following program:

```
DUMP1    CSECT
         USING DUMP1,15          ESTABLISH A BASE REGISTER
         L     1,ONE             LOAD THE FIRST NUMBER INTO R1
         L     2,TWO             LOAD THE SECOND INTO R2
         AR    1,2               ADD THE TWO NUMBERS
         ST    1,THREE           STORE THE RESULT
         XDUMP THREE,4           DUMP THE RESULT
         BCR   B'1111',14        RETURN TO CALLER
*
ONE      DC    F'64'             FIRST NUMBER
TWO      DC    F'32'             SECOND NUMBER
EOFFLAG  DC    C'0'              A FLAG SAVE AREA
THREE    DS    CL4' '            SUM OF THE TWO NUMBERS
         END   DUMP1
```

After running the above program you should be able to answer the following questions:

What is the address of the next instruction which will be executed?

What is the address of the instruction that caused the abend?

What type of error occurred?

What actually causes this error?

Correct the error by rewriting the section of code that caused it.

What is the contents of register 1 in decimal?

What does the value in reg 1 represent at the time of ABEND?

Why is the LOC address of the storage area with the label ONE on it 000018 when the branch statement before it whose LOC address is 000014 only takes up 2 bytes?

What are the contents of the two bytes of user storage starting at address 000016? What do they represent?

What are the contents of the byte saved at address 00001B? Does this byte represent the first byte of a full word?

If the dump program error were corrected, what value would the storage area at label THREE contain?

What two instructions have you worked with which cause data conversion to take place?

What is the decimal equivalent of hex `0002BA14`?

Dump Example 2

Type in and run the following program:

```
DUMP2    CSECT
         USING DUMP2,15
         LA    2,TABLE
         SR    3,3
         XREAD DATA,80
LOOP1    BM    ENDLOOP1
         XDECI 4,DATA
         ST    4,0(2,3)
         LA    3,4(,3)
         XREAD DATA,80
         B     LOOP1
ENDLOOP1 SR    3,3
         LA    7,TABLE
         LA    5,TABEND
LOOP2     CR    2,5
         BE    ENDLOOP2
         L     6,0(,2)
         ST    6,0(,7)
         L     7,4(,7)
         LA    2,4(,2)
         B     LOOP2
ENDLOOP2 BR    14
*
         LTORG
*
DATA     DS    CL80
TABLE    DC    30F'-1'
TABEND   DS    0X
         END   DUMP2
0
1  2
50
32 24 19  62
123 456 789
987 654 321
```

Using the results from the program, answer the following questions:

What was the interruption code?

What instruction caused the program to abend? Why?

What was the condition code at the time of the ABEND?

How many table entries were built? How did you figure this?

What is the return address to the calling routine? Where did you find this? Does your answer really make any sense?

What are the contents of register 7?

Was any object code changed by this program? If so, for which instructions?

Finally, explain why the program ABENDed.

Appendix C - Assembler Programming Tips

Determining Encoded Instruction Lengths

Add the following to a blank page in your Reference Summary ("Yellow Card"):

```
0 - 3    length 2
4 - B    length 4
C - F    length 6
```

This means that encoded instructions that begin with 0, 1, 2, or 3 (in hex) are 2 bytes long, those that begin with 4, 5, 6, 7, 8, 9, A or B (in hex) are 4 bytes long and those that begin with C, D, E or F (in hex) are 6 bytes long.

CSECTs always begin on a DWB (doubleword boundary).

Labeling Your Storage

By placing the following two lines in your Assembler program immediately following the LTORG at the beginning of your program's storage area, you will put a recognizable label at the beginning of that storage. Remember that, if your program dumps, the hex output is printed in two sets of four fullwords per line. More appropriately, the data is dumped on a 32-byte boundary. Also, when you look to the right of each line, you will see a display of any EBCDIC characters that happen to be found within those 32 bytes. The recognizable label will be shown there and can direct you immediately to your own storage in the dump!

```
ORG   csectname+((*-csectname+31)/32)*32
DC    C'recognizable label goes here'
```

The ORG and DC both go in column 10 and the code following each begins in column 16.

The *csectname* is, of course, the name of your program, or CSECT. Then put a recognizable character string label in between the tick marks on the second line. This character string should be no more than 32 bytes long.

LTORG

The LTORG instruction is used so that the Assembler can collect and assemble literals into what is known as a literal pool. A literal pool contains the literals specified by the programmer in an Assembly Language program. If there is an LTORG specified – and there should always be one – the literal pool is placed immediately after the LTORG. Otherwise, it is placed after the beginning of the Assembly Language program.

LTORGs always begin on a DWB (doubleword boundary).

When an LTORG is encountered, the order of movement from the Literal Table to the Symbol Table is those literals of length 8 bytes, then those of length 4, then 2 and, finally, all of the others from the longest (in bytes) to the shortest.

Addressability

Remember that the maximum displacement that can be encoded in an Assembler instruction is `X'FFF'`, or one and a half bytes of the encoded instruction itself. This maximum displace is 4095 bytes.

If a program happens to exceed 4095 bytes from its original base register of 12 as standard entry linkage dictates, an addressability error will occur in the first pass of the Assembler for those bytes that go beyond. If this happens, a second base register needs to be established. It is common to use register 11 as the second base register and, if a third is necessary, it is common to use register 10. The following establishes register 11 as a second base register that "takes over" when past `4095(,12)`:

```
         LA    11,4095(,12)
         LA    11,1(,11)
         USING csectname+4096,11
```

Of course, if a program exceeds the second base register, a third could be established. It is always best, though, to keep programs to a minimum size as they are easier to understand and debug.

Here is what the new base register establishment looks like immediately following standard entry linkage:

```
csectnme CSECT
         STM   14,12,12(13)      SAVE CALLER'S REGS
         LR    12,15             SET R12 TO R15
         USING csectnme,12       ESTABLISH R12 AS 1ST BASE REG
         LA    14,SAVEREGS       R14 -> CURRENT SAVE AREA
         ST    13,4(,14)         SAVE CALLER'S SAVE AREA ADDR
         ST    14,8(,14)         SAVE CURRENT SAVE AREA ADDR
         LR    13,14             R13 -> CURRENT SAVE AREA
*
         LA    11,4095(,12)      POINT R11 4095 BYTES BEYOND R12
         LA    11,1(,11)         NOW POINT R11 4096 BYTES BEYOND R12
         USING csectnme+4096,11  ESTABLIST R11 AS SECOND BASE REG
```

Appendix D – High Level Assembler Parameters

The High Level Assembler (HLASM) provides many assembler options for controlling the operation and output of the assembler. Default values can be set at assembler installation time for most of these assembler options. Also, a default option can be fixed so the option cannot be overridden at assembly time.

You specify the options at assembly time on:

- An external file (z/OS and CMS) or library member (z/VSE)
- The JCL PARM parameter of the EXEC statement on z/OS and z/VSE, or the ASMAHL command on CMS.
- The JCL OPTION statement On z/VSE.
- The *PROCESS assembler statement.

The assembler options are:

ADATA | NOADATA

Produce the associated data file.

ALIGN | NOALIGN

Check alignment of addresses in machine instructions and whether DC, DS, DXD, and CXD are aligned on correct boundaries.

ASA | NOASA

(z/OS and CMS) Produce the assembly listing using American National Standard printer-control characters. If NOASA is specified the assembler uses machine printer-control characters.

BATCH | NOBATCH

Specify multiple assembler source programs are in the input data set.

CODEPAGE(X'047C')

Specify the code page module to be used to convert Unicode character constants

COMPAT(*suboption*) | NOCOMPAT

Direct the assembler to remain compatible with earlier assemblers in its handling of lowercase characters in the source program, and its handling of sublists in SETC symbols, and its handling of unquoted macro operands. The LITTYPE suboption instructs the assembler to return 'U' as the type attribute for all literals.

DBCS | NODBCS

Specify that the source program contains double-byte characters.

DECK | NODECK

Produce an object module.

DXREF | NODXREF

Produce the *DSECT Cross Reference* section of the assembler listing.

ERASE | NOERASE

(CMS) Delete specified files before running the assembly.

ESD | NOESD

Produce the *External Symbol Dictionary* section of the assembler listing.

EXIT(*suboption1,suboption2,…*) | NOEXIT

Provide user exits to the assembler for input/output processing.

ADEXIT(*name(string)*) | NOADEXIT

Identify the name of a user-supplied ADATA exit module.

INEXIT(*name(string)*) | NOINEXIT

Identify the name of a user-supplied SOURCE exit module.

LIBEXIT(*name(string)*) | NOLIBEXIT

Identify the name of a user-supplied LIBRARY exit module.

OBJEXIT(*name(string)*) | NOOBJEXIT

Identify the name of a user-supplied OBJECT exit module.

PRTEXIT(*name(string)*) | NOPRTEXIT

Identify the name of a user-supplied LISTING exit module.

TRMEXIT(*name(string)*) | NOTRMEXIT

Identify the name of a user-supplied TERM exit module.

FAIL(*suboption1,suboption2,…*) | NOFAIL

MSG*(msgval)* | NOMSG

Specify the minimum severity messages which should have their severity raised.

MNOTE*(mnoteval)* | NOMNOTE

Specify the minimum severity MNOTEs which should have their severity raised.

MAXERRS*(maxerrs)* | NOMAXERRS

Specify the number of error message to be issued before assembly terminates.

FLAG(*suboption1,suboption2,…*)

Specify the level and type of error diagnostic messages to be written.

FOLD | NOFOLD

Convert lowercase characters to uppercase characters in the assembly listing.

GOFF | NOGOFF

(z/OS and CMS) Set generalized object format.

INFO | NOINFO

Display service information selected by date.

LANGUAGE(EN | ES | DE | JP | UE)

Specify the language in which assembler diagnostic messages are presented. High Level Assembler lets you select any of the following:

- English mixed case (EN)
- English uppercase (UE)
- German (DE)
- Japanese (JP)
- Spanish (ES)

When you select either of the English languages, the assembler listing headings are produced in the same case as the diagnostic messages.

When you select either the German language or the Spanish language, the assembler listing headings are produced in mixed case English.

When you select the Japanese language, the assembler listing headings are produced in uppercase English.

The assembler uses the default language for messages produced on CMS by the High Level Assembler command.

LIBMAC | NOLIBMAC

Instruct the assembler to imbed library macro definitions in the input source program.

LINECOUNT(*integer*)

Specify the number of lines to print in each page of the assembly listing.

LIST | LIST(121 | 133 | MAX) | NOLIST

(z/OS and CMS) Specify whether the assembler produces an assembly listing. The listing may be produced in 121-character format or 133-character format.

LIST | NOLIST

(VSE only) Specify whether the assembler produces an assembly listing.

MACHINE([370 | S370XA | S370ESA | S390 | S390E | ZSERIES | ZS | ZSERIES-2 | ZS-2 | ZSERIES-3 | ZS-3 | ZSERIES-4 | ZS-4 | ZSERIES-5 | ZS-5 | ZSERIES-6 | ZS-6 | ZSERIES-7 | ZS-7][,LIST | NOLIST])

Specify the operation code table to use to process machine instructions in the source program. A alternative to the OPTABLE option, the operands are also synonyms of, but are not identical to, those of the OPTABLE option.

MXREF | MXREF(FULL | SOURCE | XREF) | NOMXREF

Produce the *Macro and Copy Code Source Summary*, or the *Macro and Copy Code Cross Reference*, or both, in the assembly listing.

OBJECT | NOOBJECT

Produce an object module.

OPTABLE([DOS | ESA | UNI | XA | 370 | YOP | ZOP | ZS3 | ZS4 | ZS5 | ZS6 | ZS7][,LIST | NOLIST])

Specify the operation code table to use to process machine instructions in the source program.

PCONTROL(*suboption1,suboption2,…*) | NOPCONTROL

Specify whether the assembler should override certain PRINT statements in the source program.

PESTOP

Specify that the assembler should stop immediately if errors are detected in the invocation parameters.

PRINT | DISK | NOPRINT

(CMS) Specify that the assembler should write the LISTING file on the virtual printer.

(continued)

PROFILE | PROFILE(*name*) | NOPROFILE

Specify the name of a library member, containing assembler source statements, that is copied immediately following an ICTL statement or *PROCESS statements, or both. The library member can be specified as a default in the installation options macro ASMAOPT.

RA2 | NORA2

Specify whether the assembler is to suppress error diagnostic message ASMA066 when 2-byte relocatable address constants are defined in the source program.

RENT | NORENT

Check for possible coding violations of program reenterability.

RLD | NORLD

Produce the *Relocation Dictionary* section of the assembler listing.

RXREF

Produce the *Register Cross Reference* section of the assembler listing.

SECTALGN(*alignment*)

Specify the desired alignment for all sections, expressed as a power of 2 with a range from 8 (doubleword) to 4096 (page).

SEG | NOSEG

(CMS) Specify that assembler modules are loaded from the Logical Saved Segment (LSEG).

SIZE(*value*)

Specify the amount of virtual storage that the assembler can use for working storage.

SUPRWARN(*msgnum1,msgnum2,…*) | NOSUPRWARN

Specify one or more message numbers, of warning (4) or less severity, to be suppressed.

SYSPARM(*value*)

Specify the character string that is to be used as the value of the &SYSPARM system variable.

TERM(WIDE | NARROW) | NOTERM

Specify whether error diagnostic messages are to be written to the terminal data set On z/OS and CMS, or SYSLOG On z/VSE.

TEST | NOTEST

Specify whether special symbol table data is to be generated as part of the object module.

THREAD | NOTHREAD

Specify whether or not the location counter is to be reset at the beginning of each CSECT.

TRANSLATE(AS | *suffix*) | NOTRANSLATE

Specify whether characters contained in character (C-type) data constants (DCs) and literals should be translated using a user-supplied translation table. The suboption AS directs the assembler to use the ASCII translation table provided with High Level Assembler.

TYPECHECK(*suboption1,suboption2*) | NOTYPECHECK

Control whether or not HLASM performs type checking of machine instruction operands.

USING(*suboption1,suboption2,…*) | NOUSING

Specify the level of monitoring of USING statements required, and whether the assembler is to generate a USING map as part of the assembly listing.

WORKFILE | NOWORKFILE

If storage apart from central storage is required during assembly, use the utility file for temporary storage.

XREF(SHORT | UNREFS | FULL) | NOXREF

Produce the *Ordinary Symbol and Literal Cross Reference*, or the *Unreferenced Symbols Defined in CSECTs*, or both, in the assembly listing.