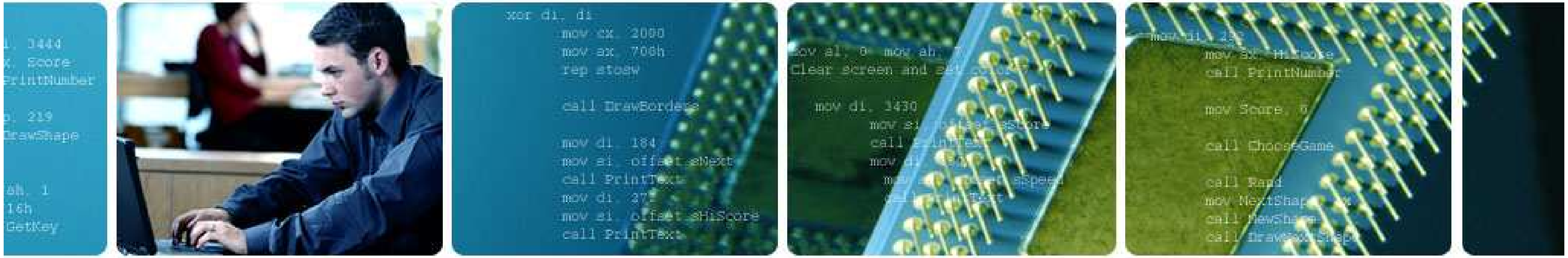
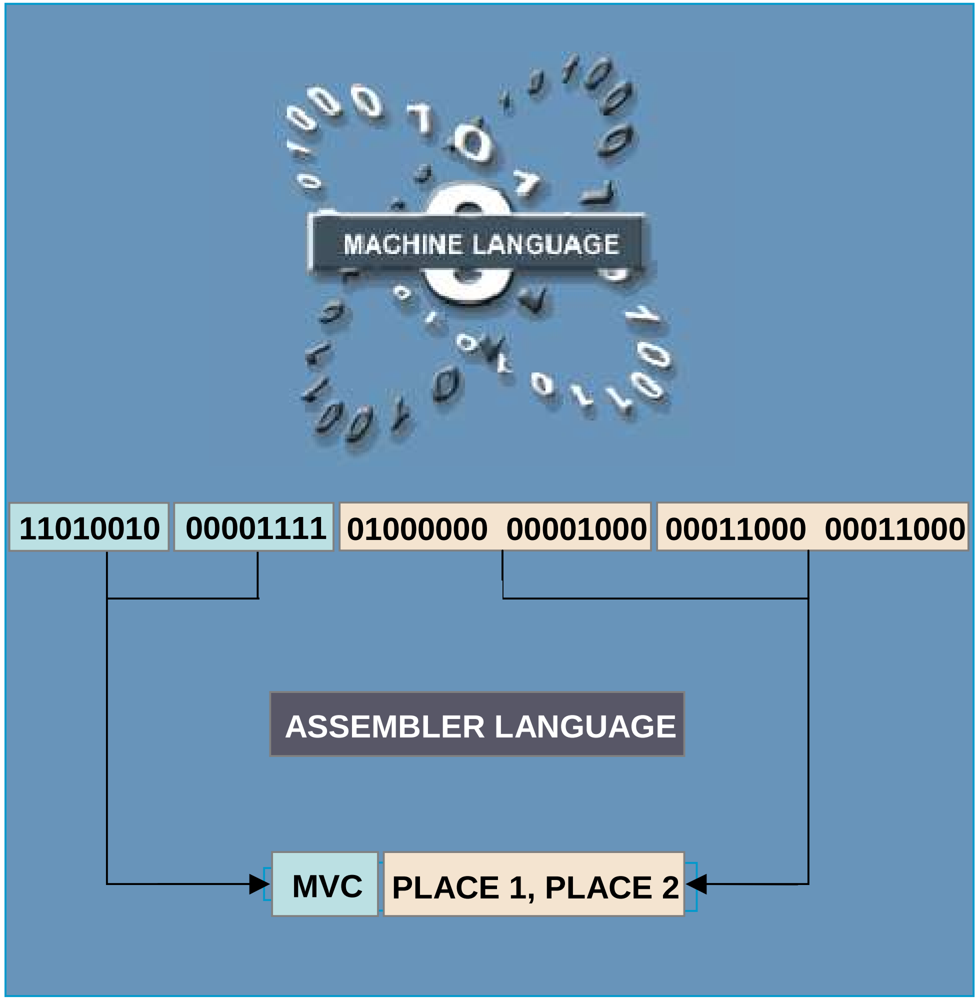
OS/390 Assembler Programming Introduction



Assemblers

What are Assemblers?

In the very early days programmers wrote in Machine Language. It was a slow, tedious and error prone process, only feasible for short and simple programs. To simplify the task of Programming and allow larger programs to be created, programs called Assemblers were created.



MACHINE LANGUAGE

11010010 00001111 01000000 00001000 00011000 00011000

ASSEMBLER LANGUAGE

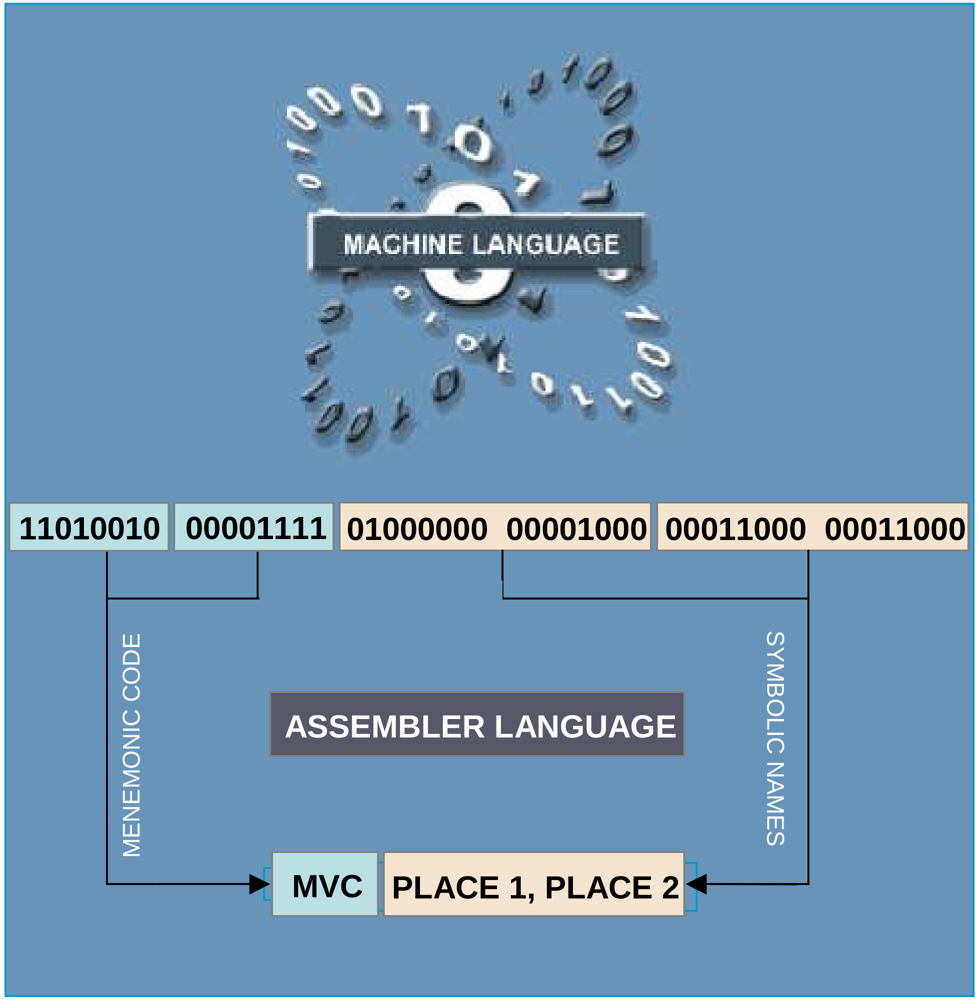
MVC PLACE 1, PLACE 2

Continued…

Assemblers (cont'd)

Assembler allowed programmers to write in Assembler Language. Although an Assembler Language program had the same number of instructions as the equivalent machine program, writing Assembler program is much simpler.

In Assembler Language, operations are represented by mnemonic codes (such as MVC for MOVE) and the data is represented by symbolic codes (such as PLACE1) rather than addresses.



Continued…

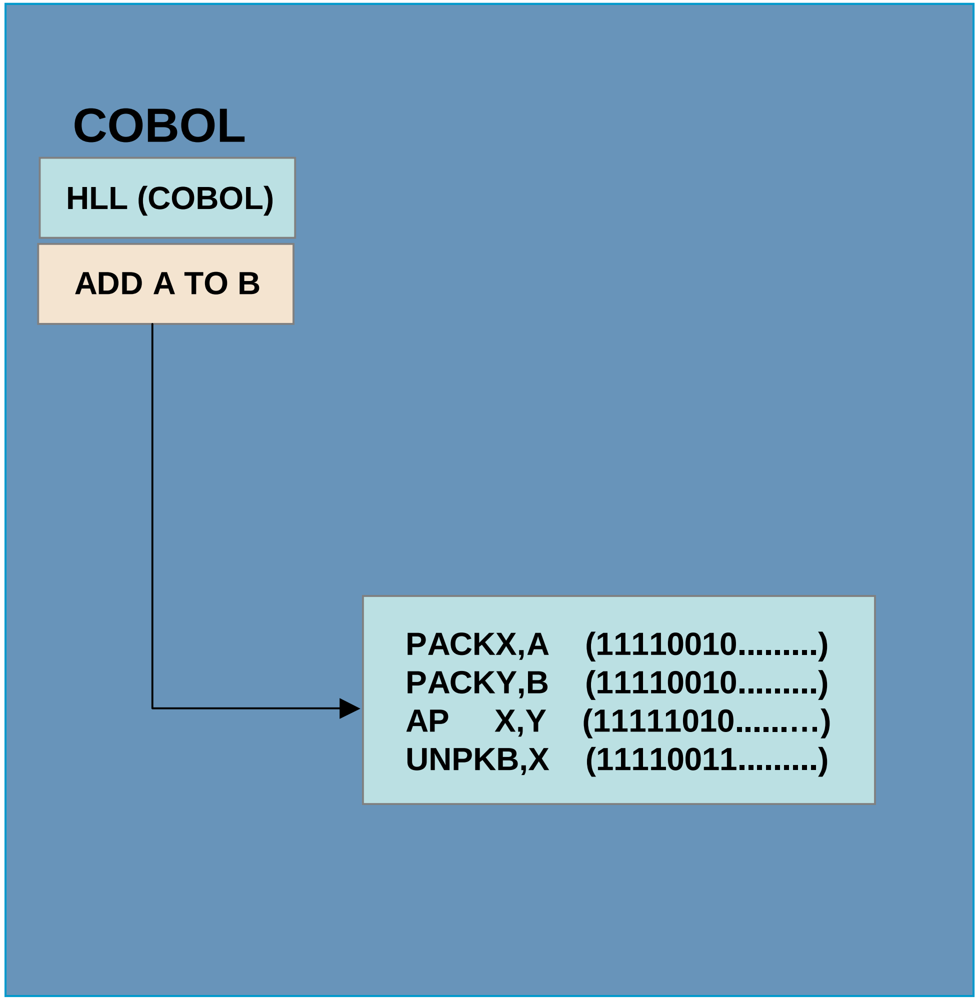
High Level Languages

What is a High Level Language?

High Level Languages (HLL) go one step further than Assembler languages in simplifying the programming task.

In a HLL, the program is expressed at a higher level of abstraction.

An Assembler language program might contain five or ten times as many instructions as the same program written in a HLL.



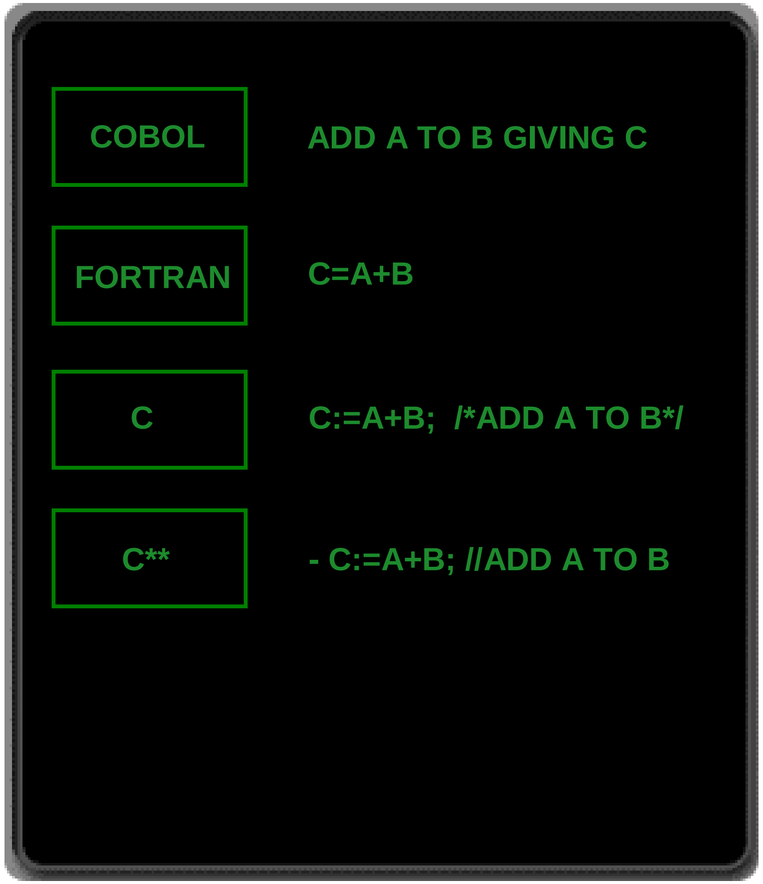
```
PACKX,A    (11110010.........)
PACKY,B    (11110010.........)
AP    X,Y  (11111010.........)
UNPKB,X    (11110011.........)
```

Continued…

High Level Languages (cont'd)

Listed below are some examples of HLL:

- Fortran – first HLL language in general use, developed for scientific and general computing
- COBOL – another early HLL, designed to facilitate the programming of business and commercial programs
- C – most popular in use today for a wide range of programming tasks
- C++ and Smalltalk – object oriented languages that are becoming increasingly popular



COBOL	ADD A TO B GIVING C
FORTRAN	C=A+B
C	C:=A+B; /*ADD A TO B*/
C**	- C:=A+B; //ADD A TO B

Continued…

Advantages of HLL and Assembler Language

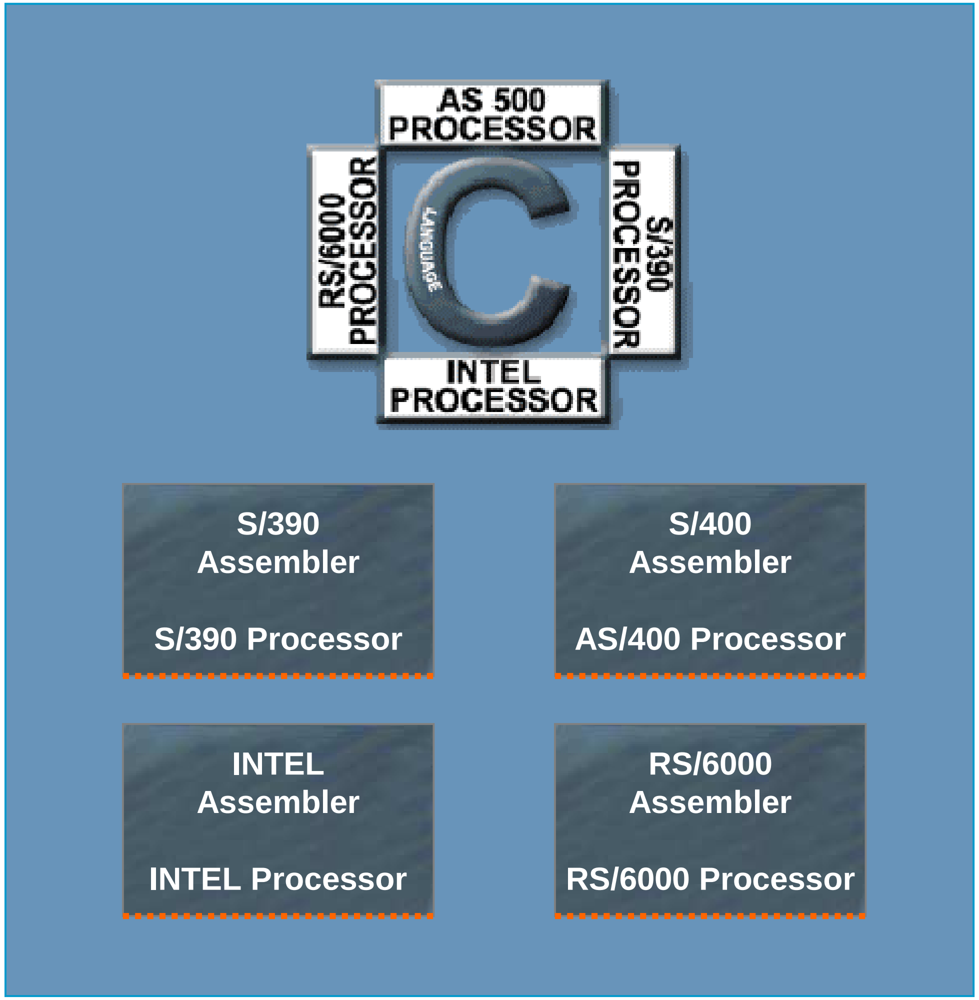
What are the advantages of HLL over Assembler Language?

HLL is advantageous over Assembler Language because of the following reasons:

- Ease of development
- Portability

What is the advantage of Assembler Language?

Assembler Language contains statements that correspond instruction by instruction to the Machine Language of the computer.



S/390 Assembler

S/390 Processor

S/400 Assembler

AS/400 Processor

INTEL Assembler

INTEL Processor

RS/6000 Assembler

RS/6000 Processor

Advantages of Assembler Language

Why program in Assembler Language?

Assembler Language is chosen for programming because of the following reasons:

- It provides extensive control of the hardware environment
- Assembler Language instructions correspond one-for-one with machine instructions making it possible to do anything the hardware allows

HLLs on the other hand, do not provide the programmer with extensive control of the hardware. The efficiency of a HLL program is subject to the way the compiler translates the HLL program.

ASSEMBLER ADVANTAGES

✓ **Control of environment**

Continued…

Advantages of Assembler Language (cont'd)

Assembler may also be considered when efficiency is a major concern.

A compiler must be designed to translate all possible valid combination of HLL code, and in some cases may not translate it to produce the most efficient Machine Language. Assembler instructions are translated one-for-one into Machine Language, and a good Assembler programmer can achieve the maximum efficiency of Machine Language code.

ASSEMBLER ADVANTAGES

✓ **Control of environment**

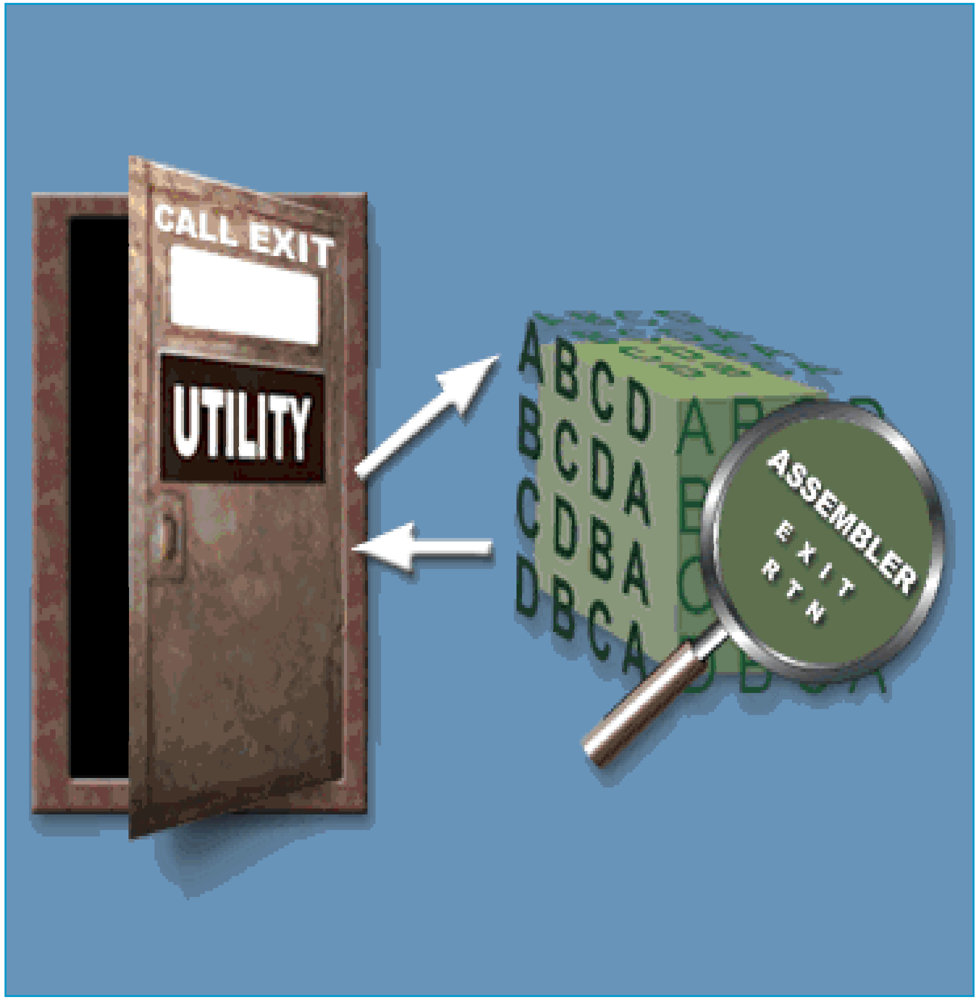
✓ **Maximum efficiency**

Continued…

Advantages of Assembler Language (cont'd)

It may not be necessary to code a whole program in Assembler Language to achieve the benefits of increased efficiency. Coding only those parts of a program which are executed frequently may achieve significant efficiency gains.

In many cases, Assembler coding is not done for a whole system or even a whole program, but only for selected modules.



Program Execution

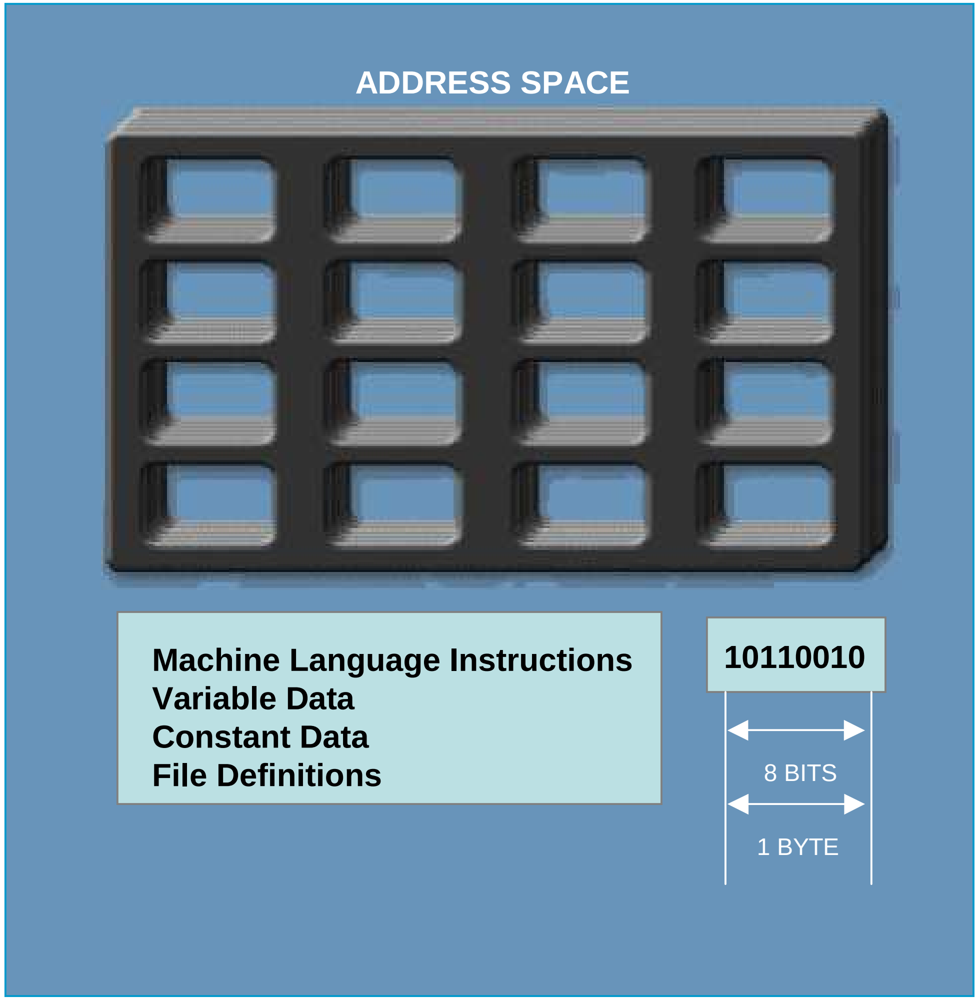
What is an Image?

In order for a program to execute, it must be loaded into the memory or main storage area of the computer. The image is a program which is loaded into the memory in order to execute in the Assembler program space.

What does the program space consist of?

A program space consists of:

- Executable Machine Language instructions
- Areas to store the program's variable data
- The program's constant data
- File definitions for use in Input/Output operations



ADDRESS SPACE

Machine Language Instructions
Variable Data
Constant Data
File Definitions

Continued…

General Purpose Registers

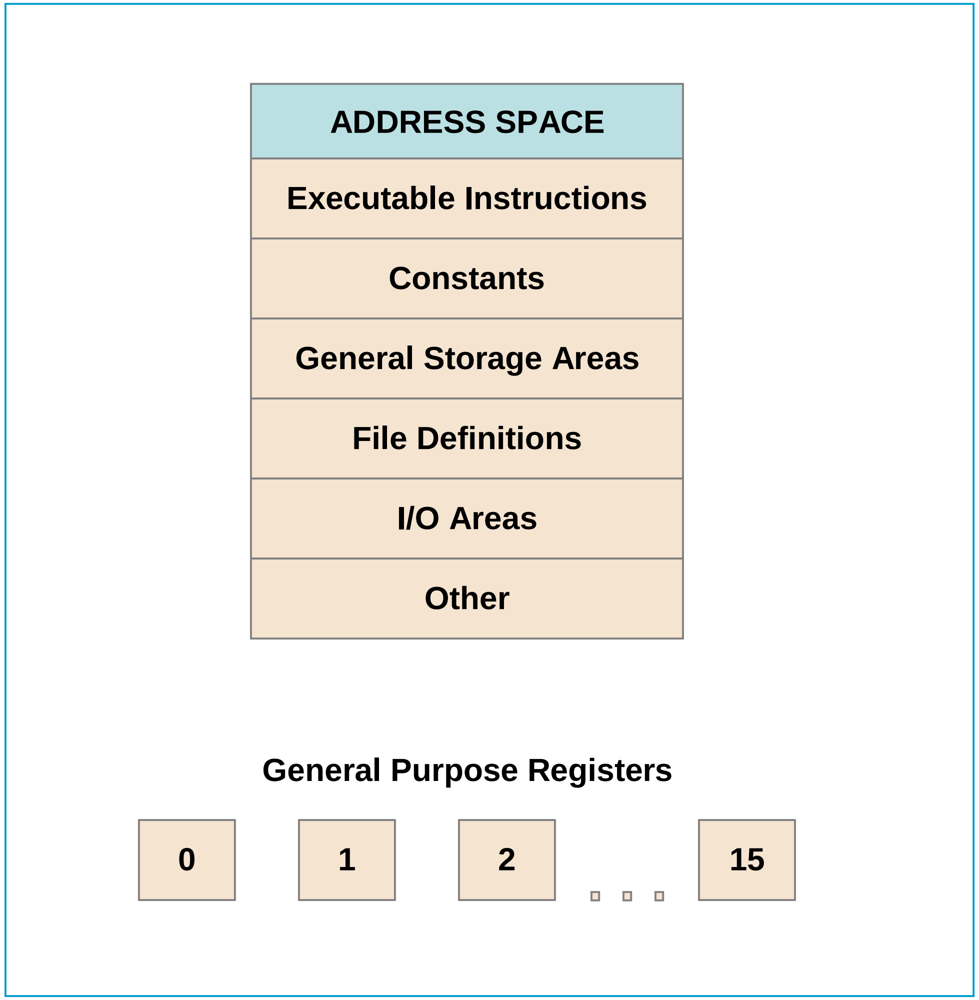
What are General Purpose Registers?

The architecture of S/390 computers provides 16 General Purpose Registers (GPRs). GPRs are part of the hardware of the computer, and can be used by any Assembler program when executing.

What are the functions of GPRs?

GPRs are very important to the Assembler programmer. They are used in addressing both instruction and data, and also for performing binary arithmetic, counting for loops and many other purposes.

GPRs are also used in conventional ways, passing control and data from one module to another.

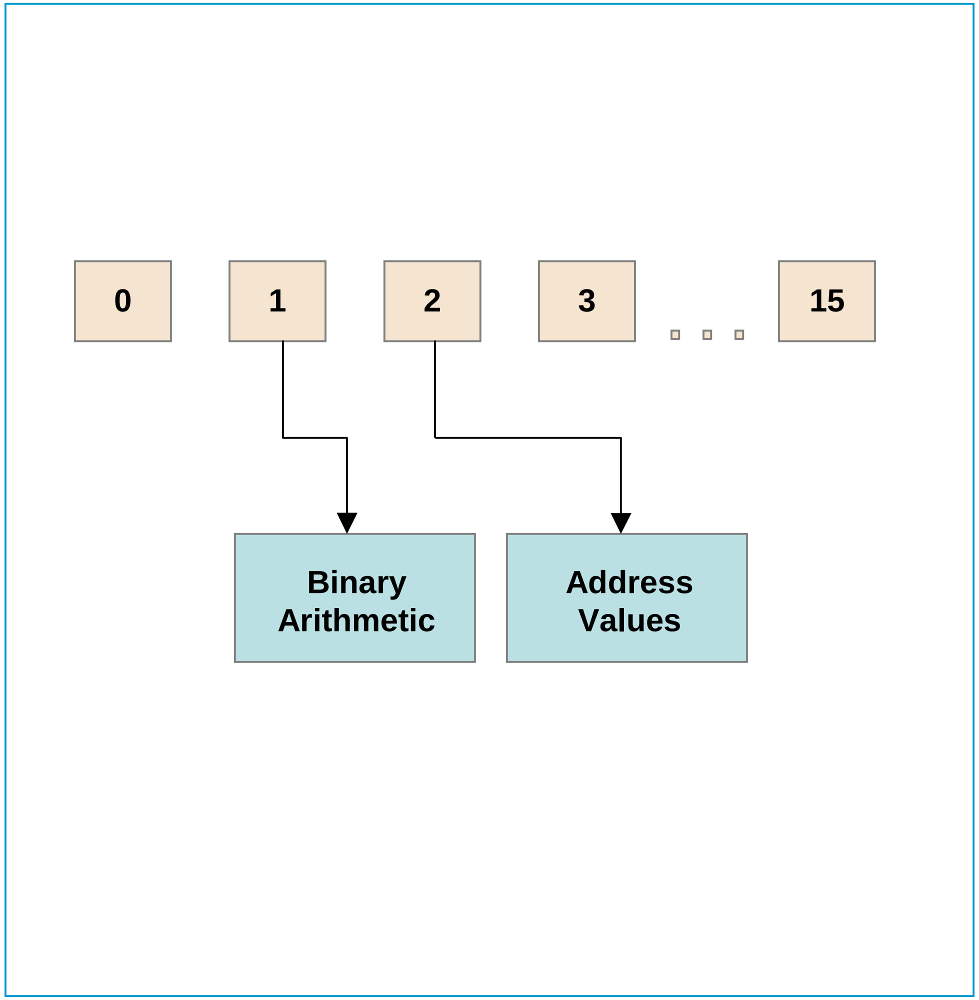


Continued…

General Purpose Registers (cont'd)

Essentials of GPRs

Effective management of GPRs is a basic skill required of the Assembler programmer. It is important to know which registers are available and for what purposes specific GPRs must be reserved.



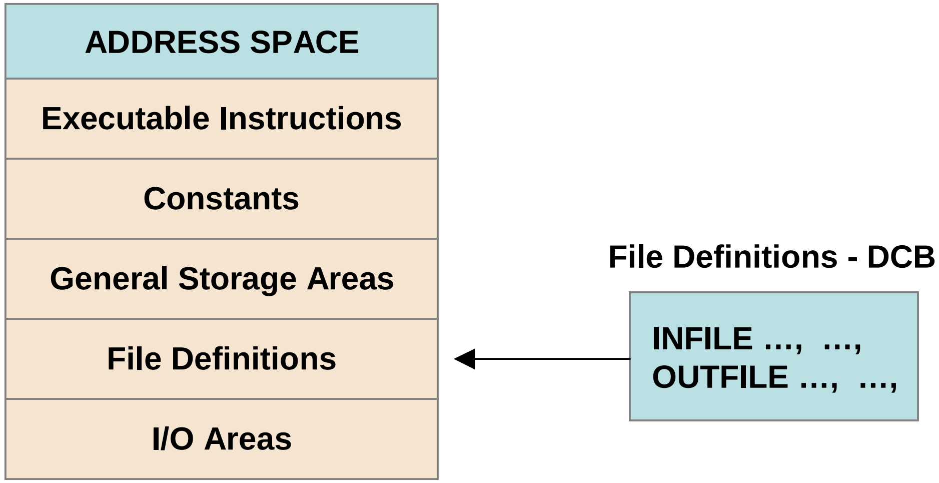
Data Control Block Definition

What is a DCB ?

A file definition, called a Data Control Block (DCB) must be created for each file the Assembler programmer wishes to access in a program. The DCB is a special kind of data area, which maintains information about an external file. The DCB is created using a system macro instruction.

The DCB contains the following:

- Name used to refer to the data
- The format and size of the records and blocks within the dataset
- Types of instructions used to access the dataset and the current status of the dataset

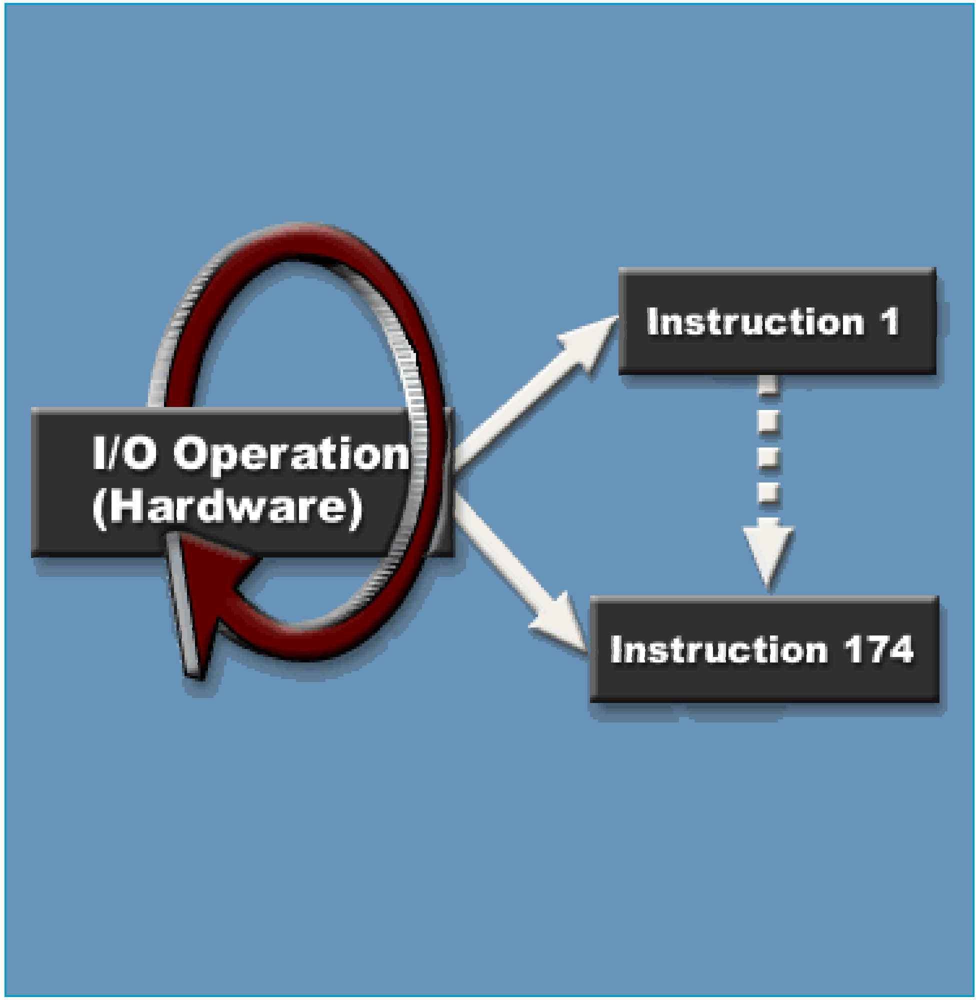


Macro Instructions

What are macro instructions?

Assembler Language instructions generally correspond one-for-one with Machine Language instructions. However there are some situations where it is not feasible for the Assembler Language programmer to code at this detailed level.

To help the programmer cope with such complex situations, the Assembler provides macro instructions. A macro instruction is an instruction that is processed by the assembler to generate a group of Assembler instructions.



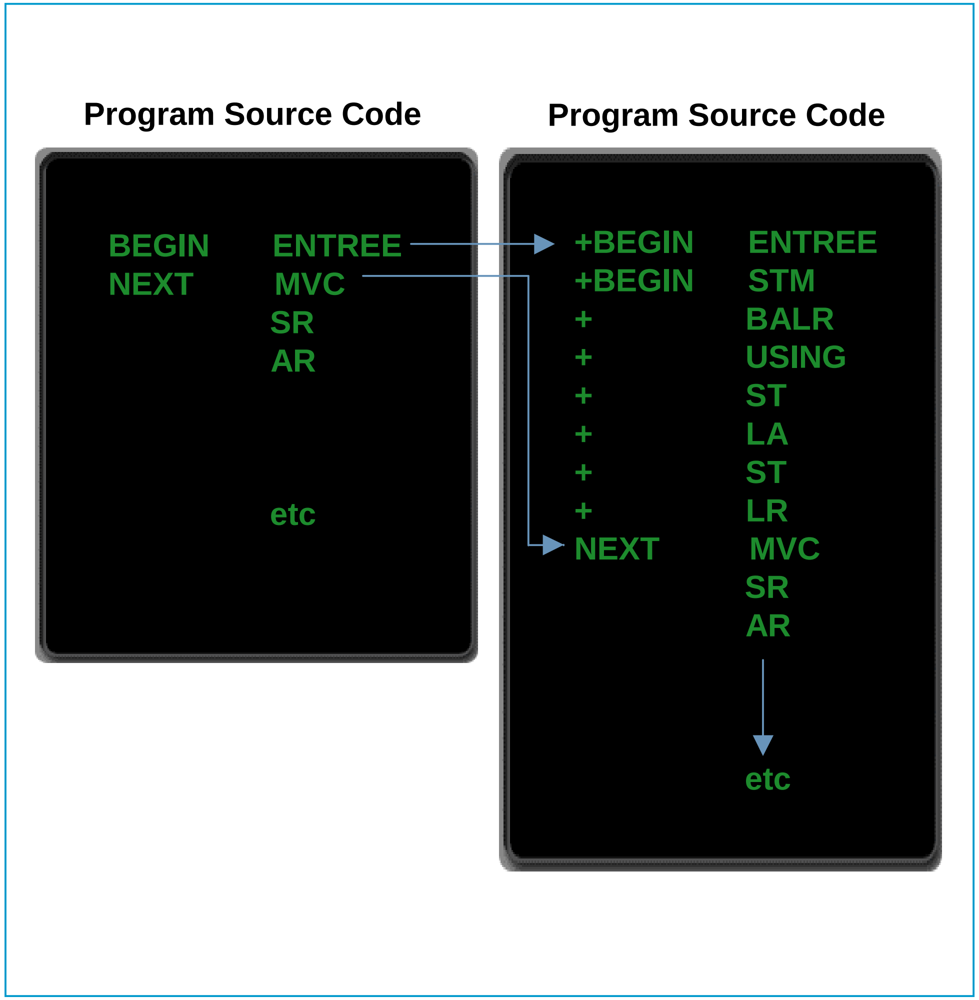
Continued…

Macro Instructions Definition (cont’d)

What are macro instructions?

The I/O macro instructions like OPEN, CLOSE, GET, PUT and ENTRÉE create multiple Assembler instructions to perform the specific I/O operation required.

The DCB macro instruction generates the various data areas and constants necessary to maintain information about an external file. The Assembler program only has to know how to handle I/O at a high level. The expansion of the macro instruction into multiple low level instructions generates the detailed level code.



Program Source Code

```
BEGIN    ENTREE
NEXT     MVC
         SR
         AR



         etc
```

Program Source Code

```
+BEGIN   ENTREE
+BEGIN   STM
+        BALR
+        USING
+        ST
+        LA
+        ST
+        LR
NEXT     MVC
         SR
         AR


         etc
```

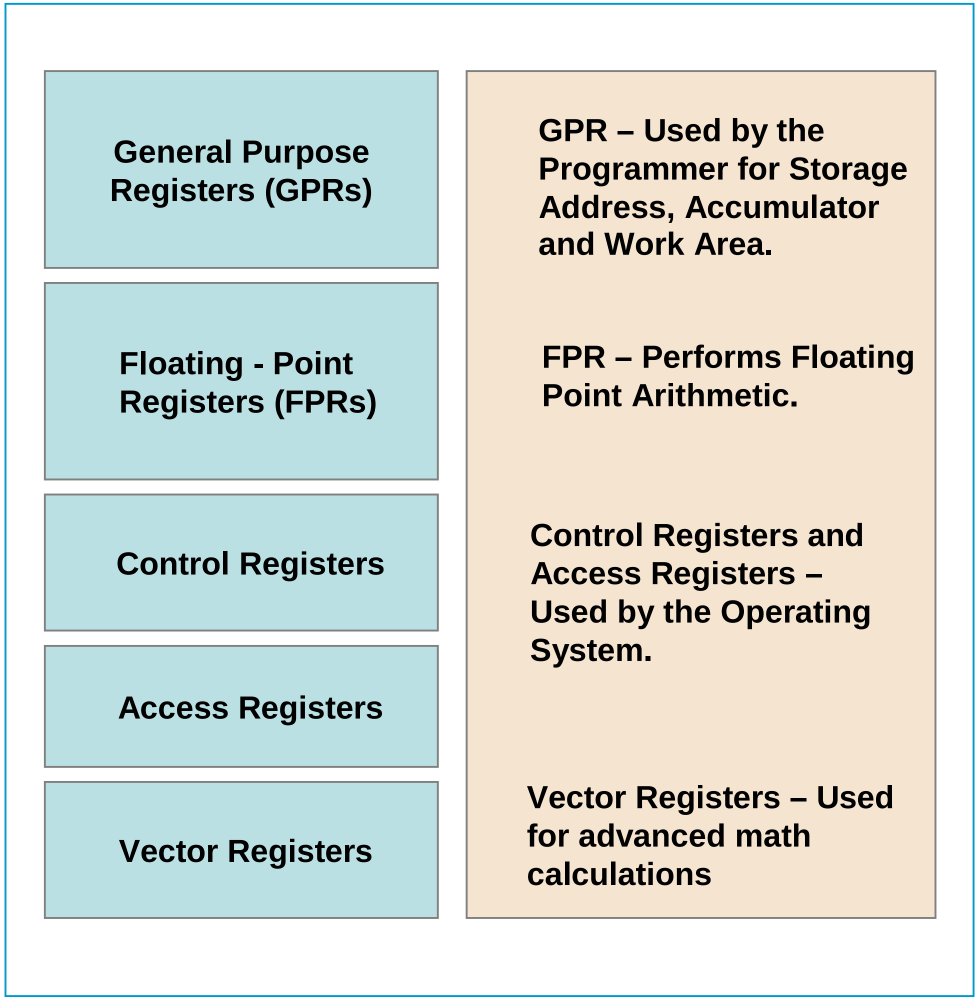
Registers

What are registers?

Registers are another location where operands reside in an Assembler Language program. There are several registers in S/390 architecture.

Types of registers in S/390 architecture are:

- General Purpose Registers (GPRs)
- Floating Point Registers (FPRs)
- Control Registers
- Access Registers
- Vector Registers



General Purpose Registers (GPRs)

Floating - Point Registers (FPRs)

Control Registers

Access Registers

Vector Registers

GPR – Used by the Programmer for Storage Address, Accumulator and Work Area.

FPR – Performs Floating Point Arithmetic.

Control Registers and Access Registers – Used by the Operating System.

Vector Registers – Used for advanced math calculations

General Purpose Registers

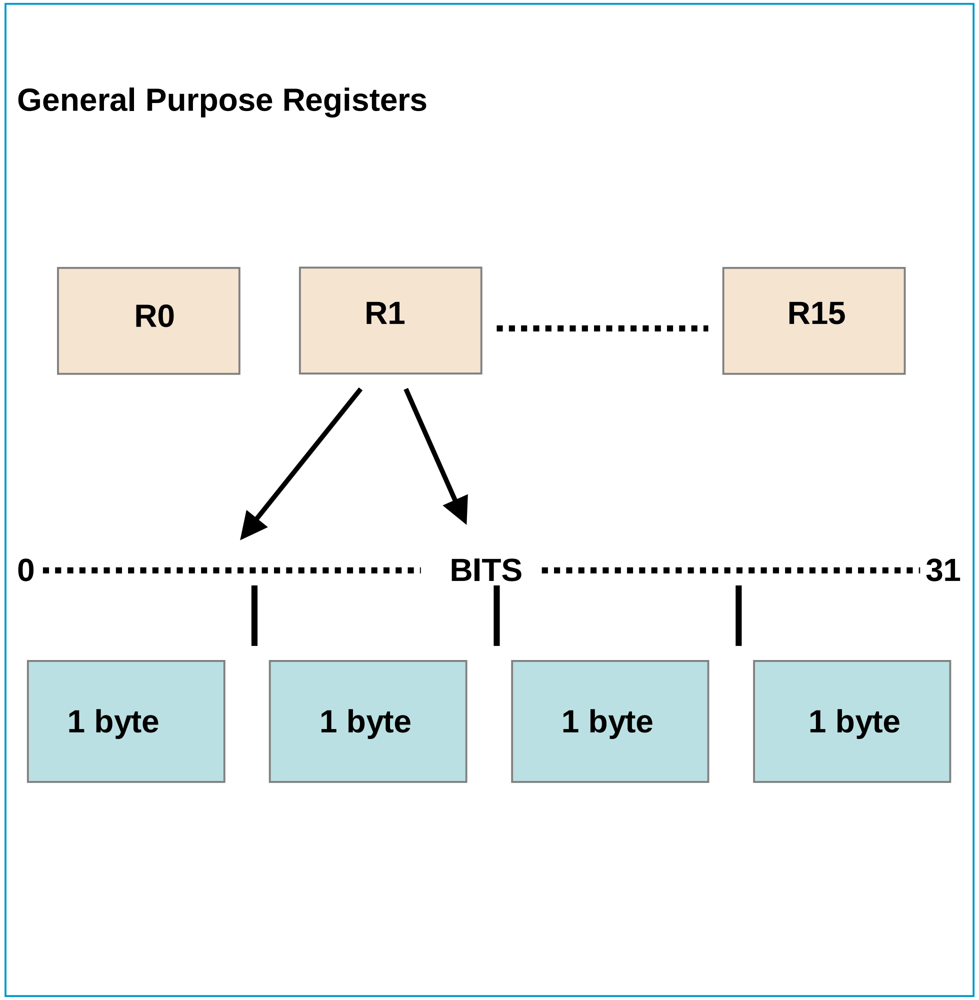
What are General Purpose Registers (GPRs)?

GPRs are of the most interest to programmers. There are 16 general purpose registers (GPRs), numbered from 0-15. They are referred to with a prefix of R (R0, R1, R2…. R15).

The GPRs are 32 bits in length and can hold fixed-point values between -2, 147, 483, 648, and +2, 147, 483, 647.

GPRs are used as:

- Part of the address for all storage operands
- As accumulators and work areas in performing fixed-point arithmetic



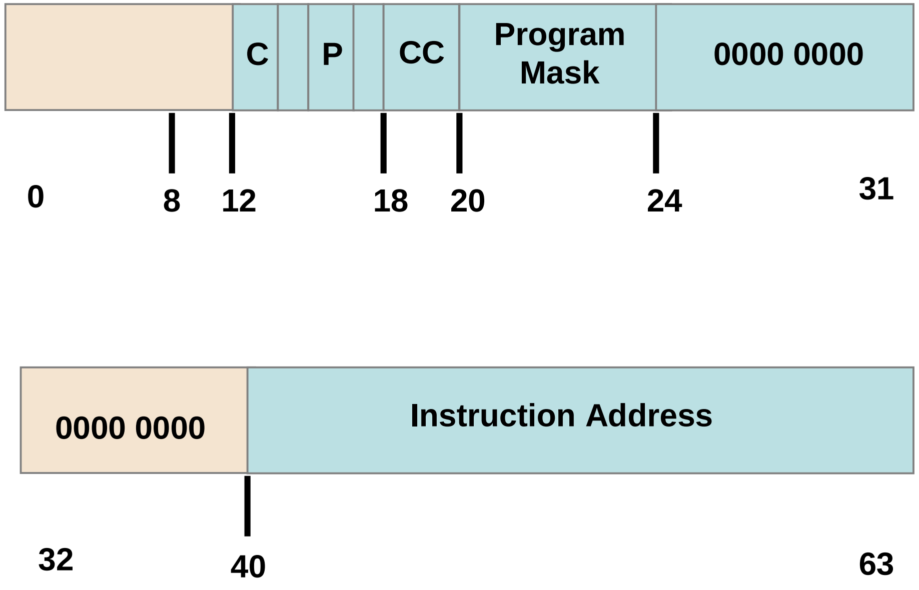
Program Status Word

What is a Program Status Word (PSW)?

The Program Status Word is a hardware location where the current status of the Central Processing Unit (CPU) is represented. It is a register that is of particular interest to the Assembler Language programmer.

Amongst other fields, the PSW contains:

- The address of the next instruction to be executed
- The current value of the condition code (CC)
- Other flags, such as the one to indicate whether the CPU is in problem state or supervisor state



Program Status Word (EC Mode)

Arithmetic Instructions

Although EBCDIC represents numbers in character form, it cannot be used for arithmetic operations.

What are the kinds of arithmetic instructions?

Fixed-Point (Binary) arithmetic and Decimal arithmetic are integer based, handling only whole numbers . Floating-Point arithmetic handles real numbers, numbers with fractional portions.

Fixed Point (Binary)	**0111111111111111**
Decimal: **Zoned Decimal** **Packed Decimal**	**F3F2F7F6C7** **32767C**
Floating Point:	**278.31924**

Note!

Briefly look at each of the types and how it is used to represent numeric values.

Fixed-Point Arithmetic

What is Fixed-Point arithmetic?

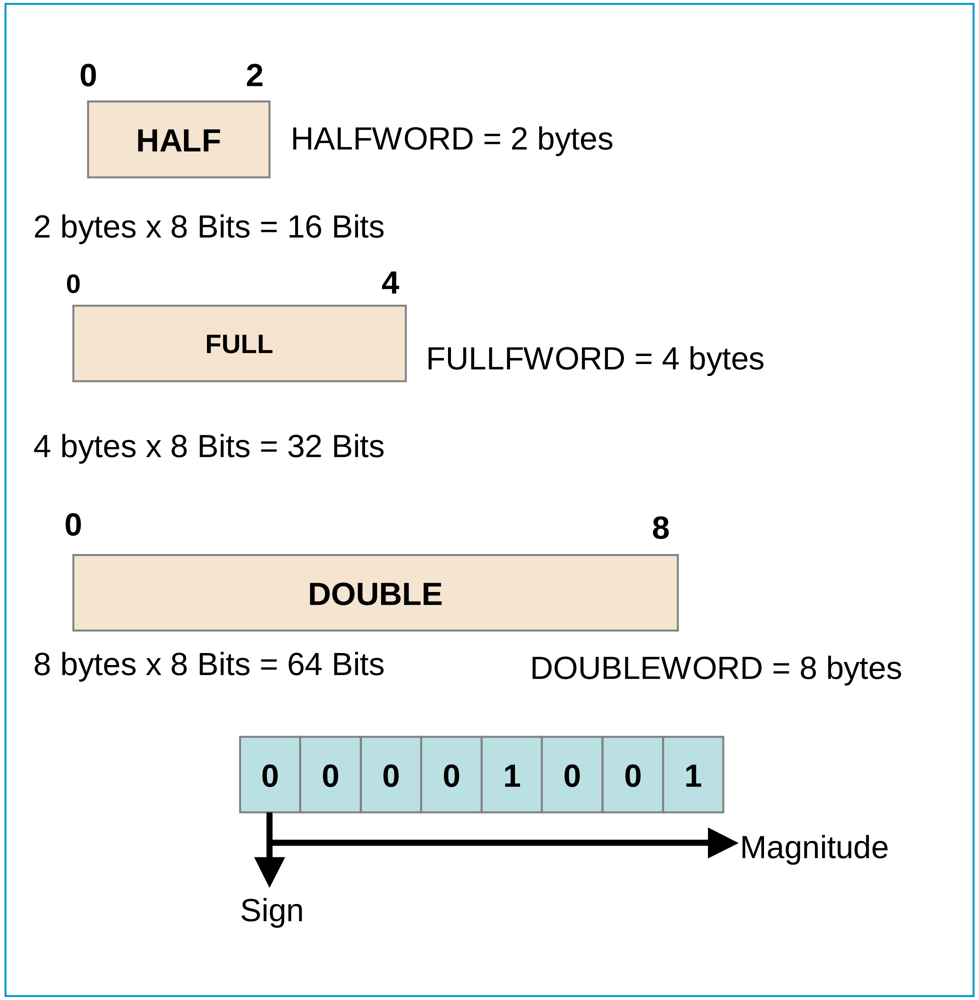
Fixed-Point arithmetic uses three data types:

- Full words – sometimes called a word, is four bytes (32 bits) in length
- Half words – 2 bytes (16 bits) in length
- Double words – 8 bytes (64 bits) in length

What do fixed-point numbers consist of?

Fixed-point numbers consist of two parts, a sign and a magnitude. The sign is in the leftmost, or high-order bit, and the magnitude occupies the rest of the field.

A zero bit represents a positive sign, a one bit a negative sign.



0 2

HALF

HALFWORD = 2 bytes

2 bytes x 8 Bits = 16 Bits

0 4

FULL

FULLFWORD = 4 bytes

4 bytes x 8 Bits = 32 Bits

0 8

DOUBLE

8 bytes x 8 Bits = 64 Bits

DOUBLEWORD = 8 bytes

0	0	0	0	1	0	0	1

Magnitude

Sign

Continued…

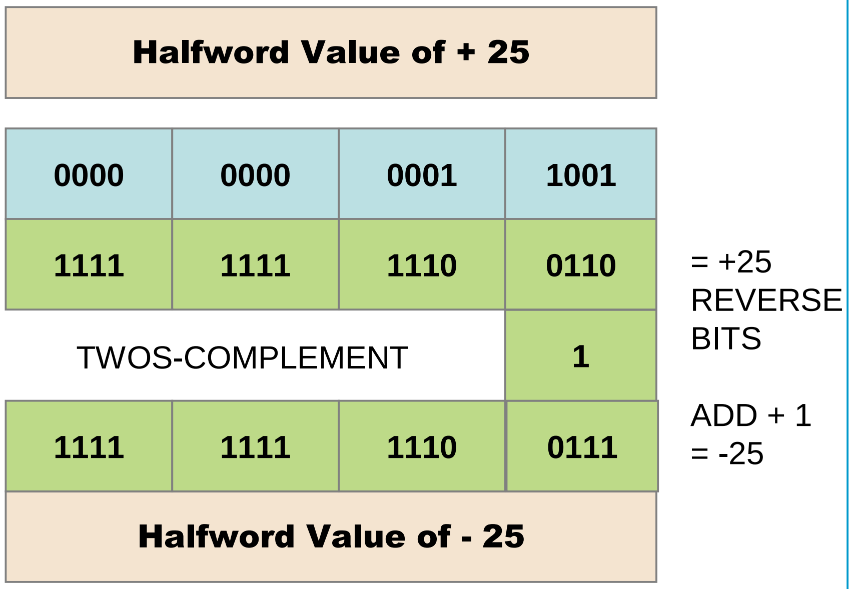
Fixed-Point Arithmetic (cont'd)

For positive numbers, the magnitude is represented in true binary. A half word value of +25 would be represented as:

0000 0000 0001 1001

For negative numbers the magnitude is expressed in twos-complement form. To convert a binary value to twos-complement, flip all the bits, change 1s to 0 and 0s to 1, and then add one. A halfword value of -25 would be represented as:

1111 1111 1110 0111



Halfword Value of + 25

0000	0000	0001	1001	= +25
1111	1111	1110	0110	REVERSE BITS
TWOS-COMPLEMENT			1	
1111	1111	1110	0111	ADD + 1 = -25

Halfword Value of - 25

Unit: S/390 Memory Usage Topic: Data Representation

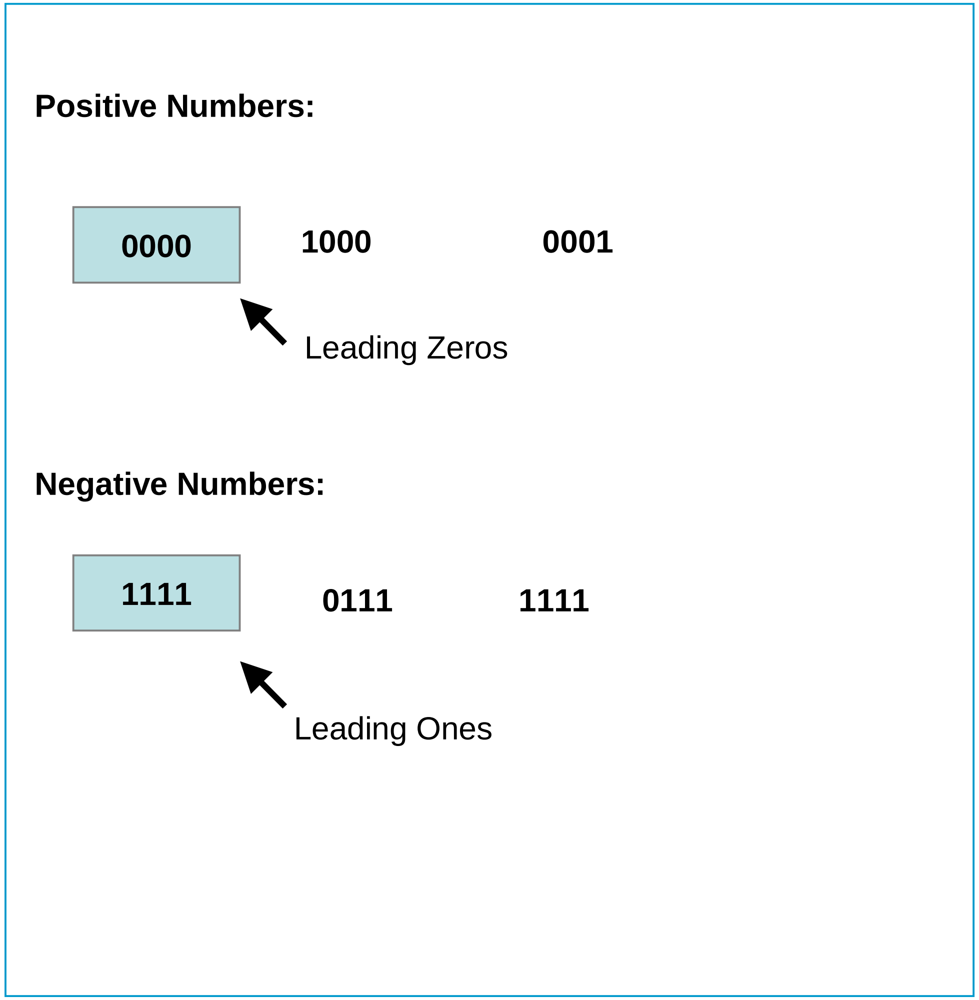
Leading Zeros and Twos-complement

Leading zeroes

Note that in positive numbers, leading zeroes (0s) are insignificant, and in negative numbers, leading one (1s) are insignificant. In general, bits equal to the sign bit are insignificant.

Twos-complement

Twos-complement notation used to represent negative fixed-point numbers makes doing arithmetic easy for the computer.



Types of Decimal Numbers

Decimal numbers are of the following two types:

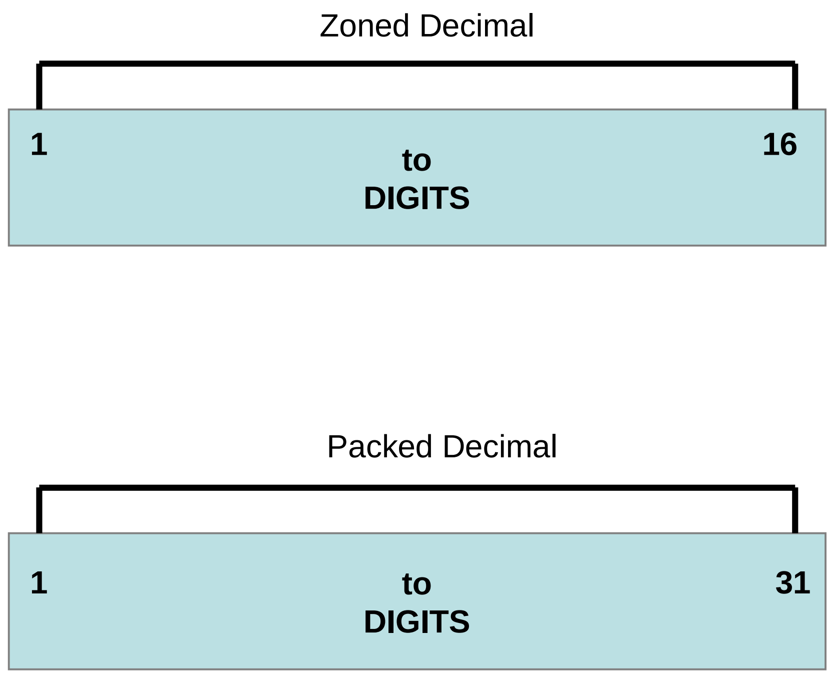
- Zoned Decimal
- Packed Decimal

What is packed decimal?

Packed decimal requires less storage than EBCDIC, since each number is represented by half a byte (4 bits). Arithmetic is only done on packed decimal numbers.

What is zoned decimal?

Zoned decimal is an intermediate format, used in converting numbers from characters to packed decimal. Packed numbers can be from 1 to 31 digits in length, while zoned can be from 1 to 16 digits.



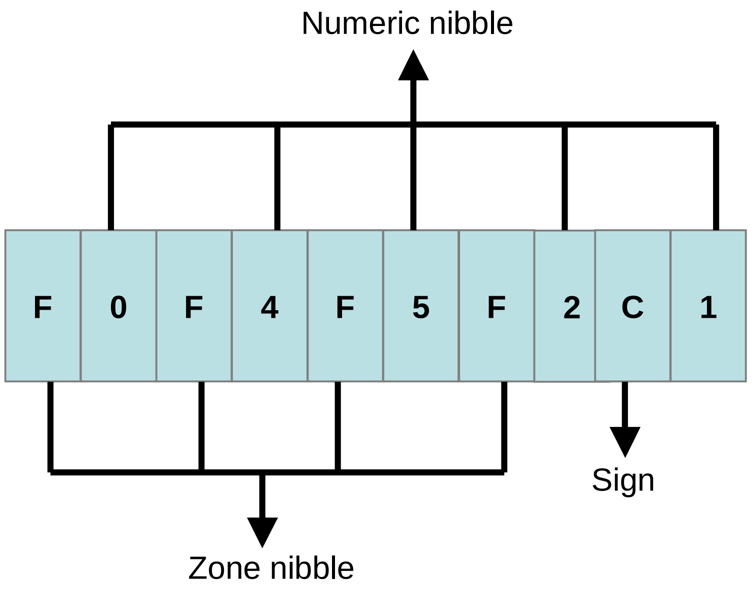
Zoned Decimal Number

What is a zone nibble?

Each byte in a zoned decimal number consists of a zone nibble and a numeric nibble. The zone nibble is the first half of each byte, and the numeric nibble is the second half.

Each numeric nibble contains the decimal digit value for that position. All zone nibbles except the rightmost contain the hex value F (one one binary). The rightmost nibble contains the sign. The hex character C represents a positive sign and D represents a negative sign.

Zoned Decimal Number:

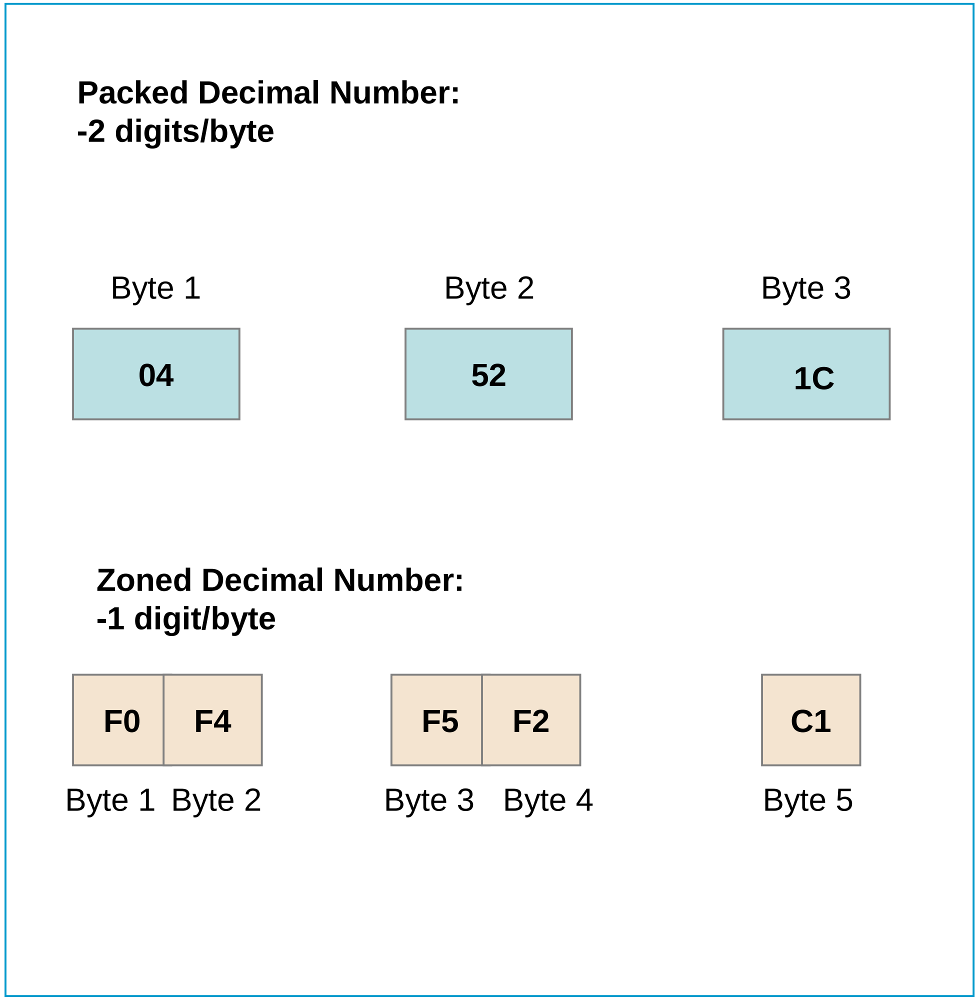


Note! A nibble is half a byte.

Packed Decimal Number

Packed decimal notation adds one decimal digit into each nibble, except for the rightmost nibble, which contains the sign. The sign is coded the same way as for zoned decimal.

A three byte packed decimal number containing the value 4521 would be represented (in hex) as: 04521C



Packed Decimal Number:
-2 digits/byte

Byte 1	Byte 2	Byte 3
04	**52**	**1C**

Zoned Decimal Number:
-1 digit/byte

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
F0	**F4**	**F5**	**F2**	**C1**

Using Floating Point Numbers

Fixed point and decimal numbers are integers. Real numbers, numbers with a fractional part, are represented with floating point numbers. Floating point numbers allow you to represent a large range of numbers.

$1.74642 \text{ X } 10^{17}$

= 174642000000000000

Floating-Point Numbers

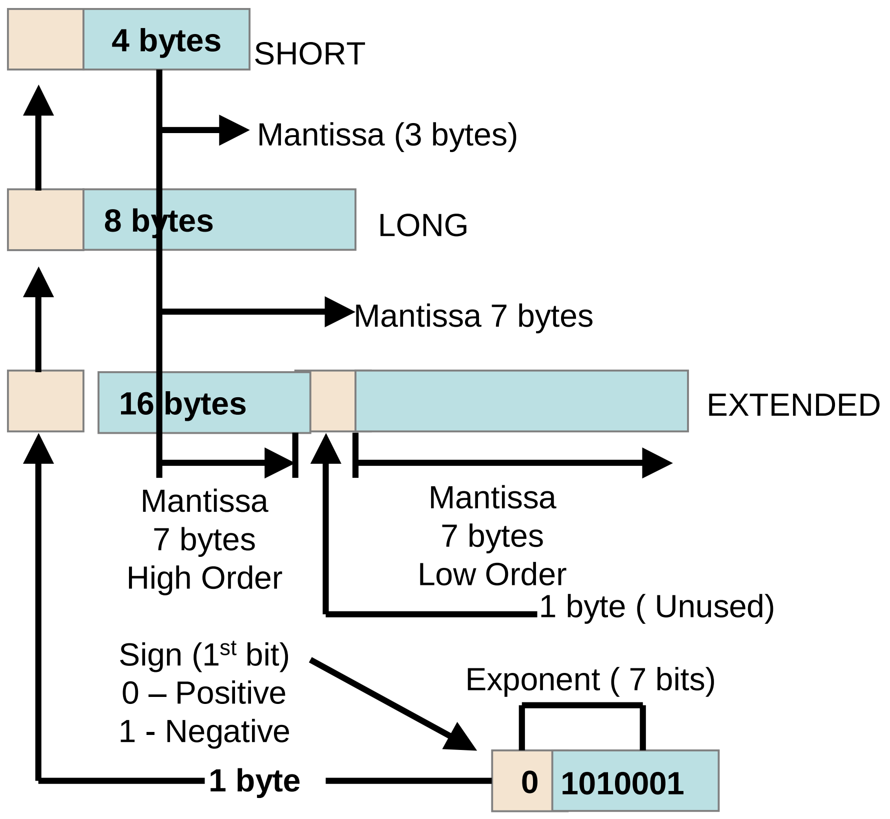
Following are the 3 data types used for floating-point numbers?

- Short numbers – 4 bytes in length
- Long - 8 bytes in length
- Extended – 16 bytes in length

A floating-point number consists of the following parts:

- A Sign
- A Mantissa
- An Exponent

Floating Point Numbers:

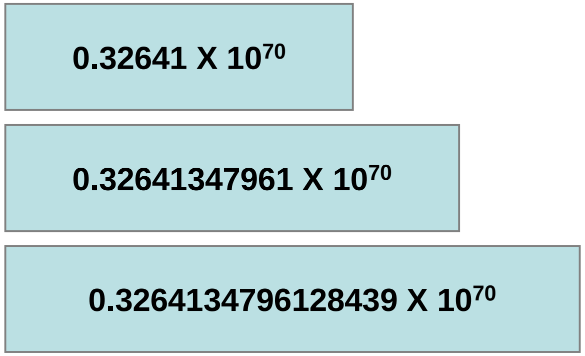


Floating-Point Formats

The three floating-point formats all provide for essentially identical ranges of magnitude, since the exponent is the same size in each.

Going from short to long or from long to extended simply increases the length of the mantissa, and thus provides more precision.

Floating-Point Formats



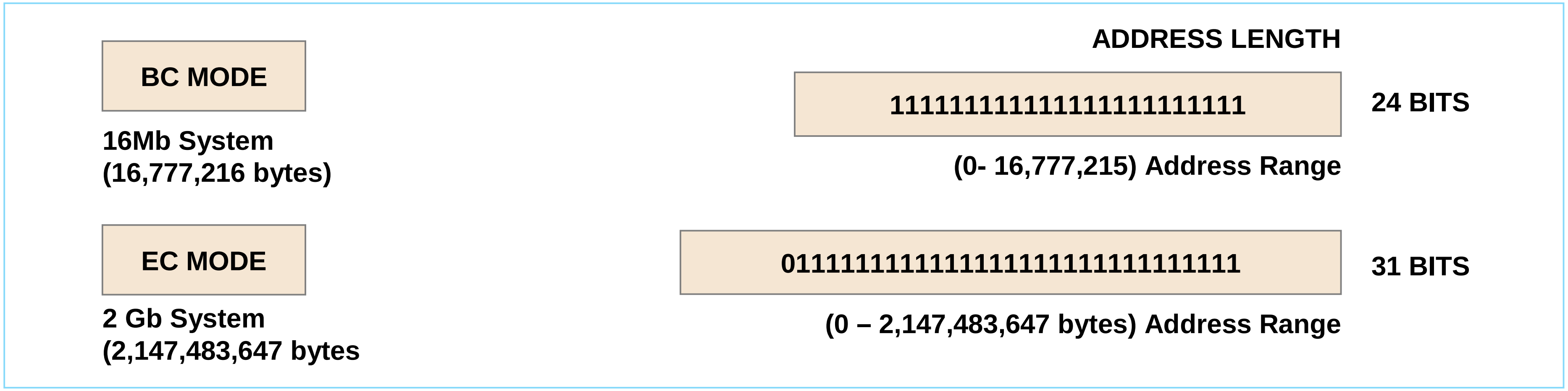
0.32641×10^{70}

$0.32641347961 \times 10^{70}$

$0.3264134796128439 \times 10^{70}$

Unit: S/390 Addressing and Instructions Topic: Addressing

Modes of Operation



ADDRESS LENGTH

BC MODE

16Mb System
(16,777,216 bytes)

111111111111111111111111

24 BITS

(0- 16,777,215) Address Range

EC MODE

2 Gb System
(2,147,483,647 bytes

0111111111111111111111111111111

31 BITS

(0 – 2,147,483,647 bytes) Address Range

What are the two modes of operation in System /390 architecture?

One of the most distinctive features of a computer’s architecture is the way it addresses main storage. The S/390 architecture provides two modes of operation:

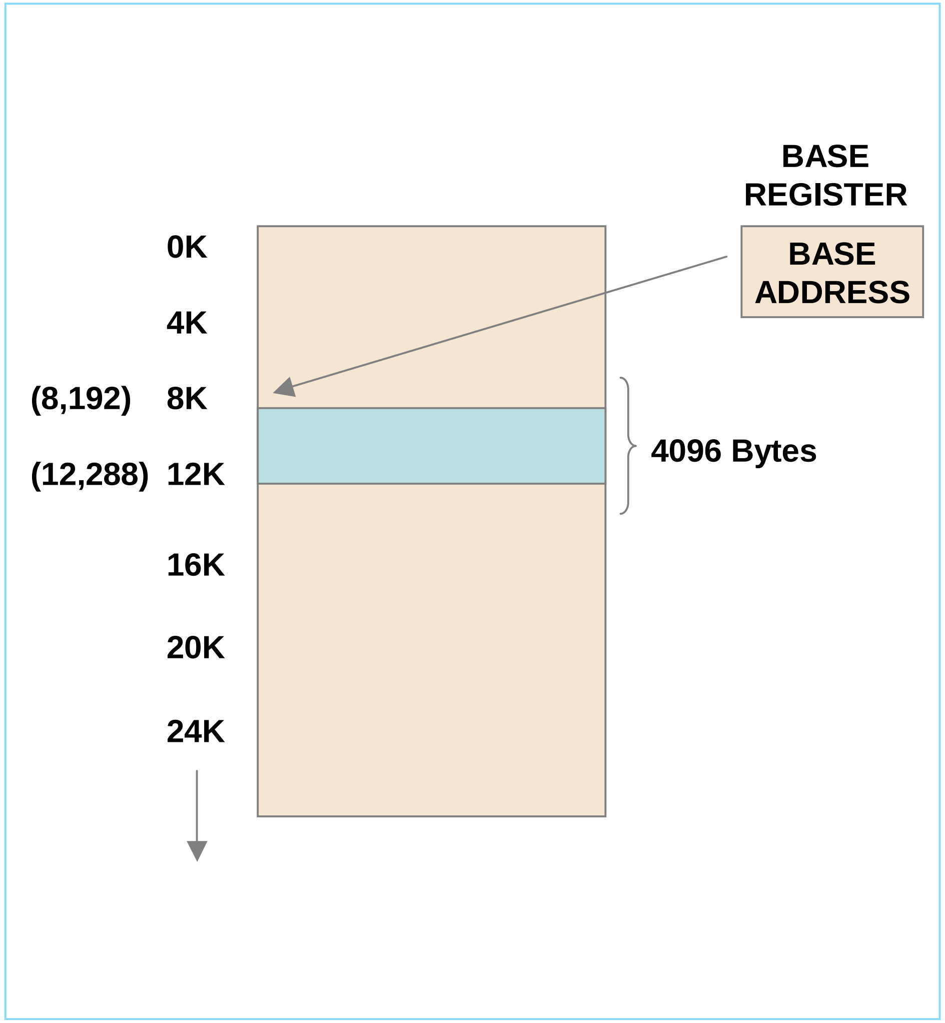
- Basic Control Mode (BC)
- Extended Control Mode (EC)

Unit: S/390 Addressing and Instructions Topic: Addressing

Address Space

Flat address space means that the memory is continuously addressable, with no segmentation of memory from the programmer's point of view.

In a few cases the programmer must deal with absolute addresses. However the most common concern of a programmer is addresses within instructions.

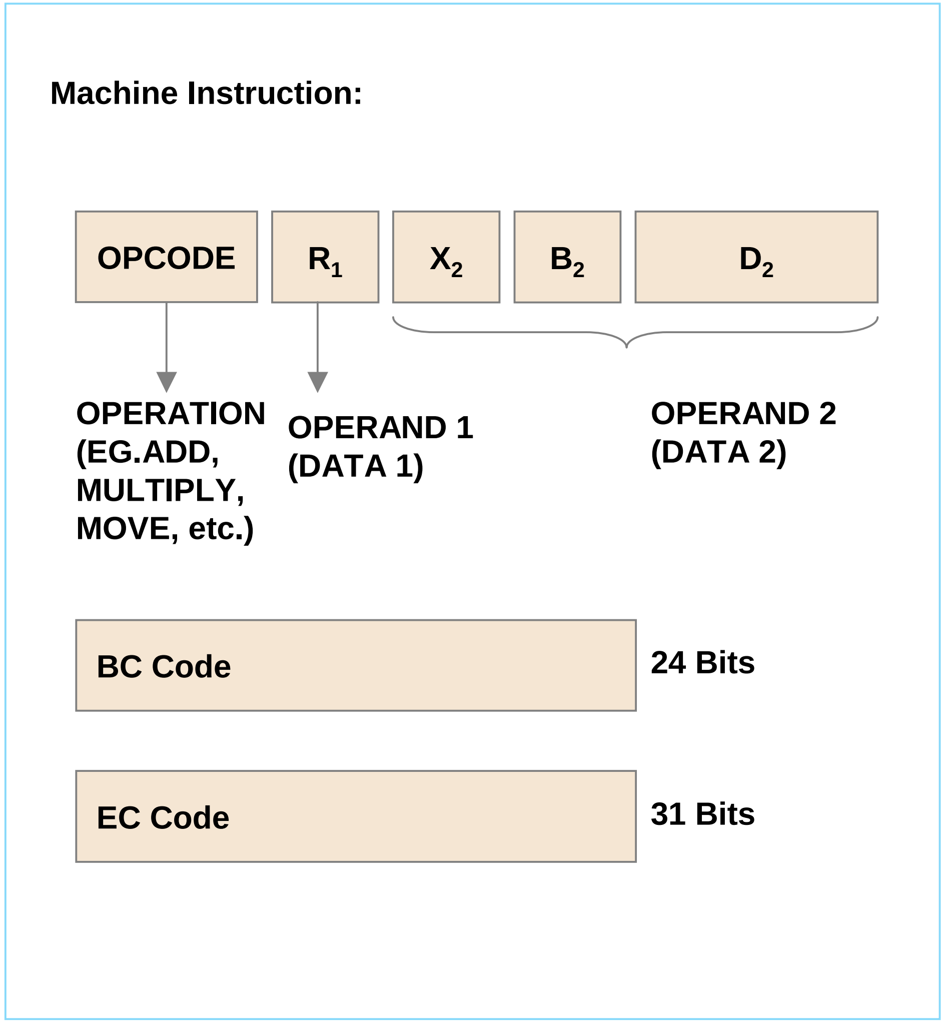


Machine Instruction

Machine instructions consist of operations (add, multiply and so on); and a number of operands, specifying the data upon which the operation is to be performed. The operands may specify registers or memory locations.

Architecture with instructions specifying absolute addresses is generally not designed because:

- The length of absolute addresses depends on the mode BC or EC.
- If absolute addresses were used in instructions, either instructions would vary in length depending on the mode, or there would be wasted space when BC mode was used. Specifying 31 bits for each address in an instruction will produce very long instructions.
- Using absolute addresses in instructions affects program relocatability.

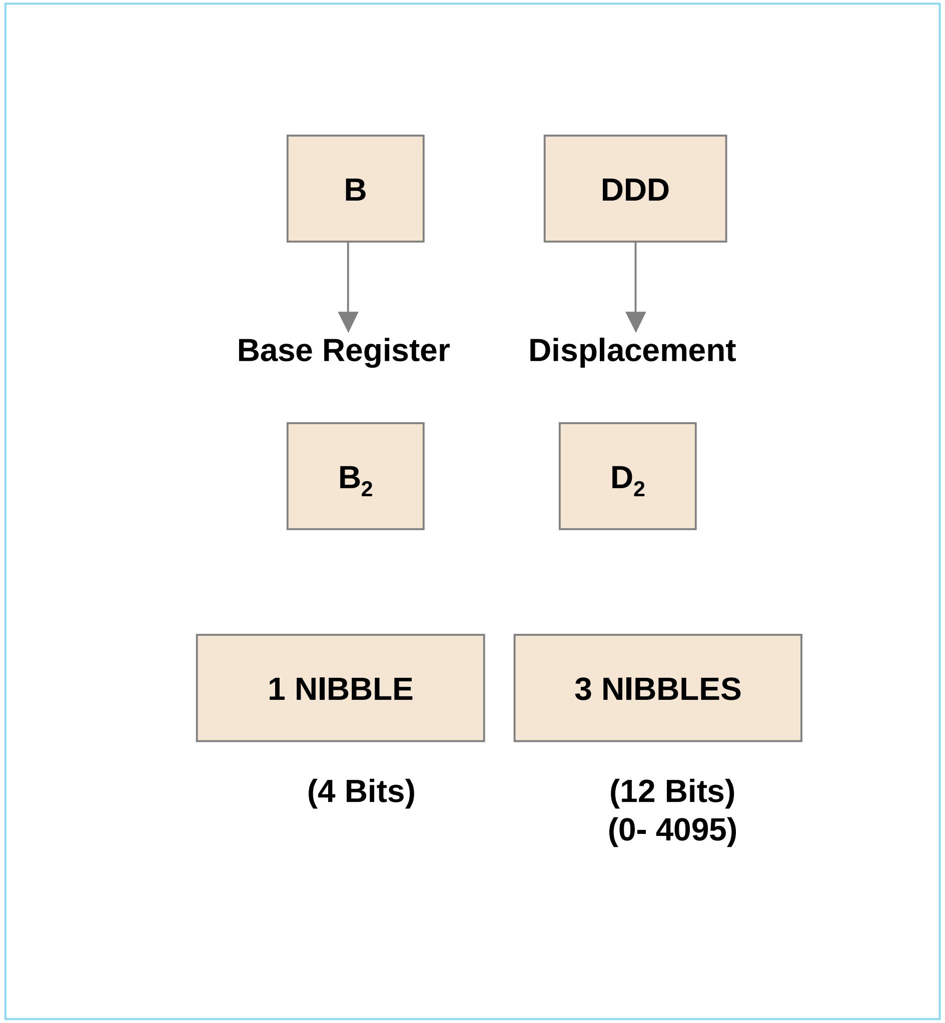


Base Displacement Address

What is Base Displacement Addressing?

The form of address used in instructions is called base displacement addressing. In this form the address is specified as a displacement from the address contained in a GPR. This is represented as a BDDD. B stands for base, and when a GPR is used in this way it is called a base register.

There are 16 GPRs, numbered 0 – 15, so any register can be represented by one hex digit (one nibble). The displacement DDD is a 3 nibble or a 12 bit quantity, and so contains values from 0 – 4095. When the processor is decoding an instruction, it converts each base-displacement address to an absolute address by adding the specified displacement DDD to the contents of the specified base register B.



Continued…

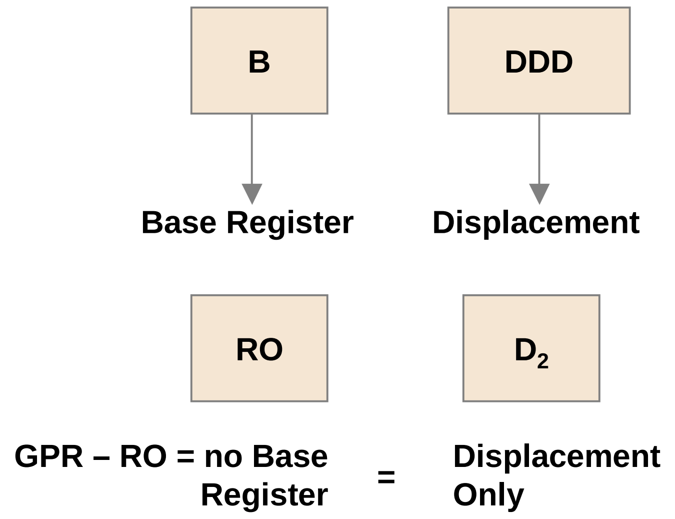
Base Displacement Address (cont'd)

There is an exception to this rule of address formation. If the base register B specifies register 0, it means that no base register is to be used in the address calculation. That is, you cannot use R0 as a base register in the normal way, but to specify that you wish to use displacement only in the address calculation.

What are RX instructions?

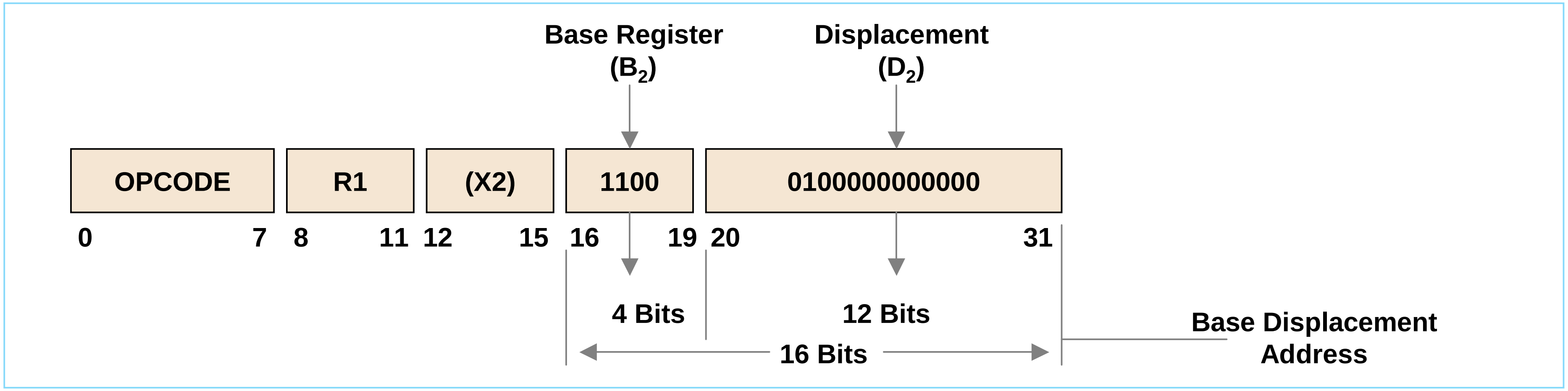
It will be seen in the next section unit 2 -1, that there is one instruction type which extends this address formation process. RX instructions use two registers, a base register B and an index register X , along with a displacement in address generation.

Note! With RX instructions, if R0 is specified as base and/or index register, it is ignored in the address calculation.



Continued…

Base Displacement Address (cont'd)



A base displacement address uses 16 bits, 4 to specify the base register, and 12 to specify the displacement.

When it decodes the instruction, prior to execution, the processor takes the contents of the base register (32 bits), adds the displacement and drops either 1 high order bit to get a 31 bit EC address, or 8 high order bits to get a 24 bit BC address. The instructions are kept short, while still specifying full address indirectly.

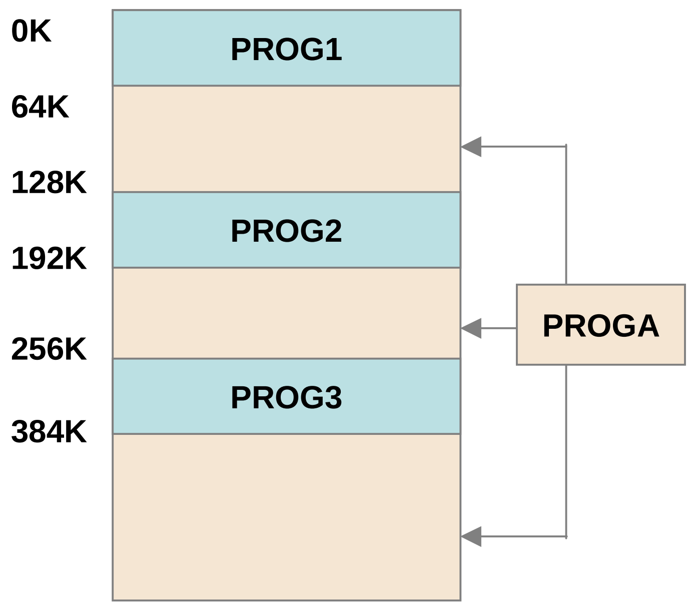
Advantages of Base Displacement

Advantages of base displacement are:

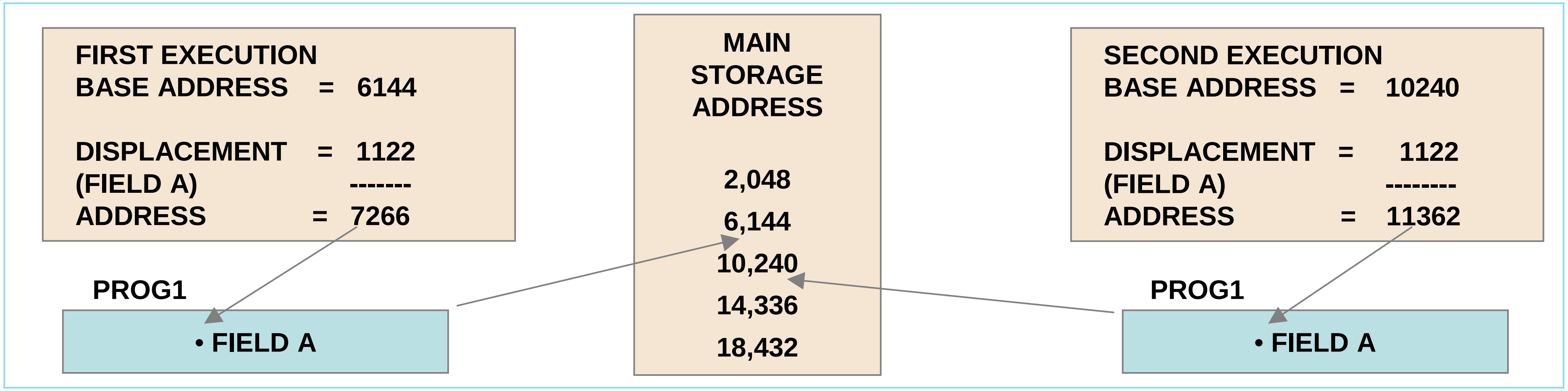
- Base displacement addresses keep instruction length short
- It allows for common instruction format with no wasted space, regardless of processor mode
- Provides easy relocatability of programs

Why is relocatability important?

It is highly desirable, from an operating system point of view, that programs be allowed to run on any location in main storage. If instructions used absolute addresses, they would always have to be loaded in the same storage location in order to run properly.



Absolute Addresses

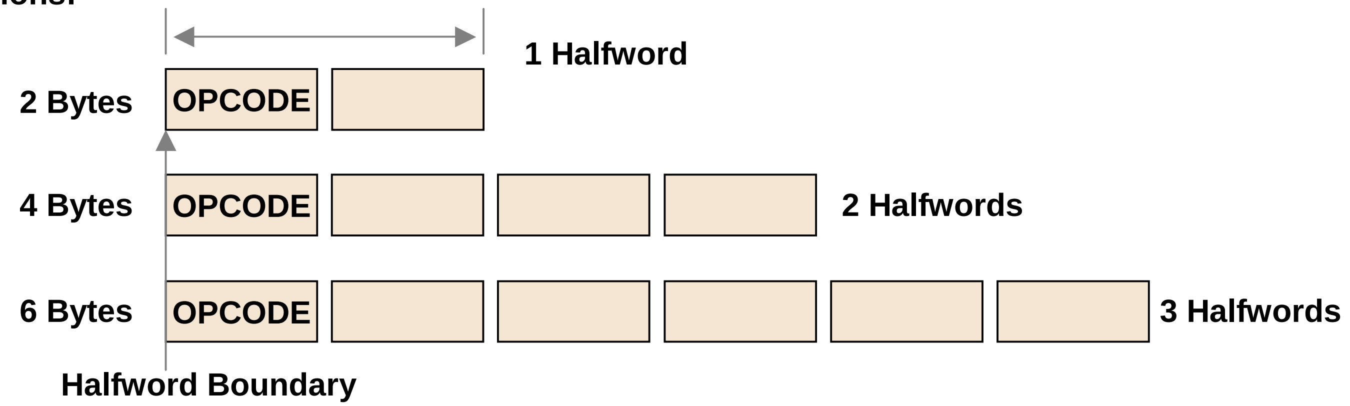


Many programs do contain some absolute addresses. These addresses do have to be adjusted when the program is loaded into main storage for execution to reflect the program's actual load point. This is a relatively simple task, since the number of absolute addresses is usually very small.

Unit: S/390 Addressing and Instructions Topic: Machine Instruction Formats

Assembler Language Instructions

Assembler Language Instructions:



Assembler Language instructions are 1,2 or 3 halfwords in length, and are always aligned on a halfword boundary, which is an address evenly divisible by 2.

What are opcodes?

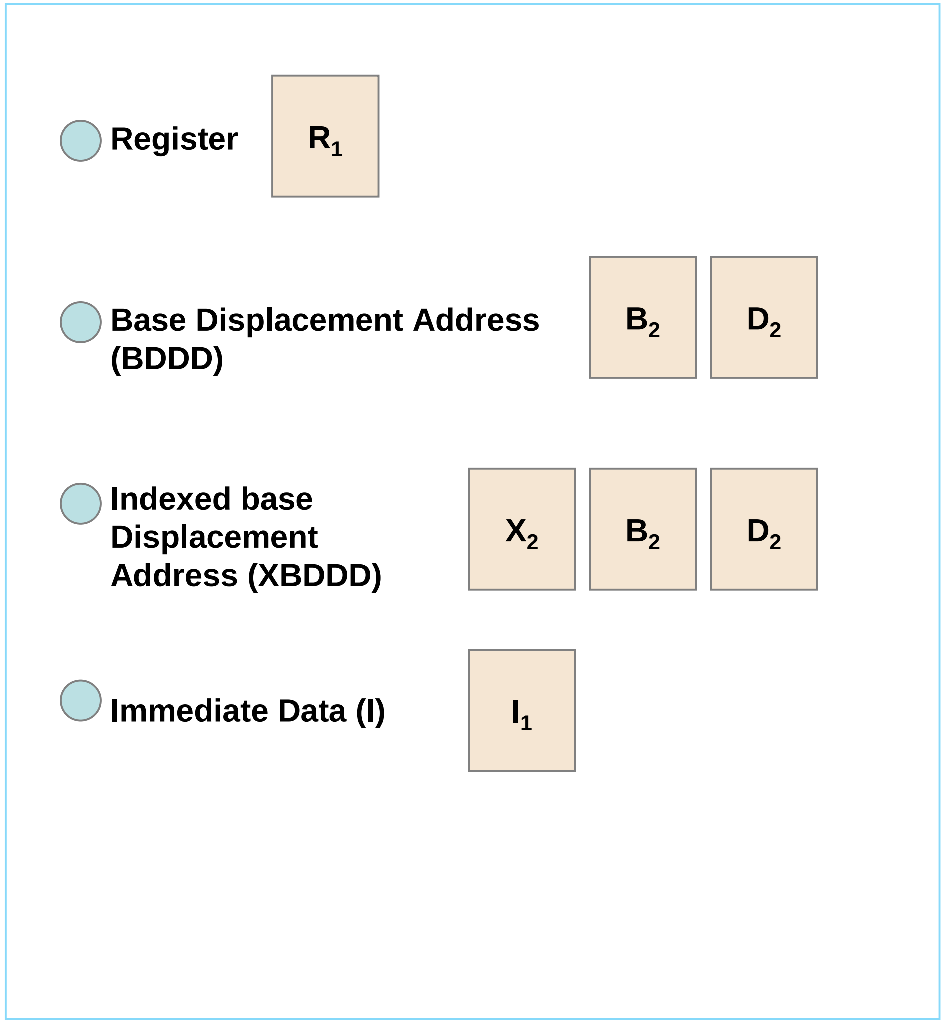
The first field in every instruction is the operation code (opcode), which is 1 byte in length (there are two instructions with 2 byte opcodes, but they are not commonly used in application programming). The operation code specifies what the instruction does, such as add, divide, move or compare.

Types of Operands

What are the different types of operands?

Most instructions have 2 operands, but some have 0, 1, 2 or 3. The different types of operands are:

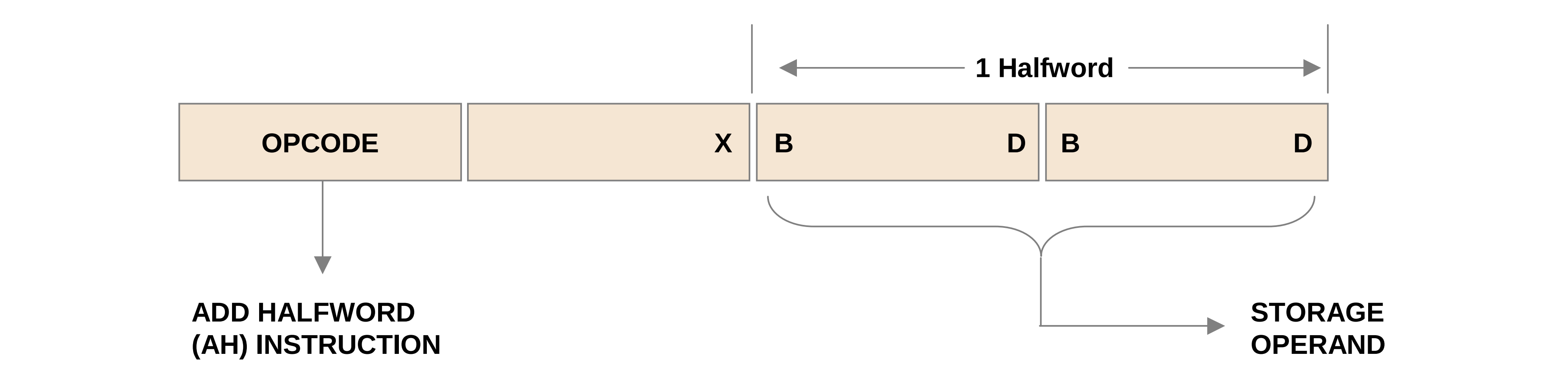
- Registers, specified by a register number
- Main storage location, specified by a base displacement address (BDDD)
- Main storage locations, specified with an indexed base displacement address (XBDDD)
- A single byte of immediate data, contained within the instruction itself (I).



Register
R_1
Base Displacement Address (BDDD)
B_2
D_2
Indexed base Displacement Address (XBDDD)
X_2
B_2
D_2
Immediate Data (I)
I_1

Continued…

Types of Operands (cont’d)



In some cases, the length of a storage operand is implied by the instruction. The storage operand for an Add Halfword (AH) instruction is a halfword in length.

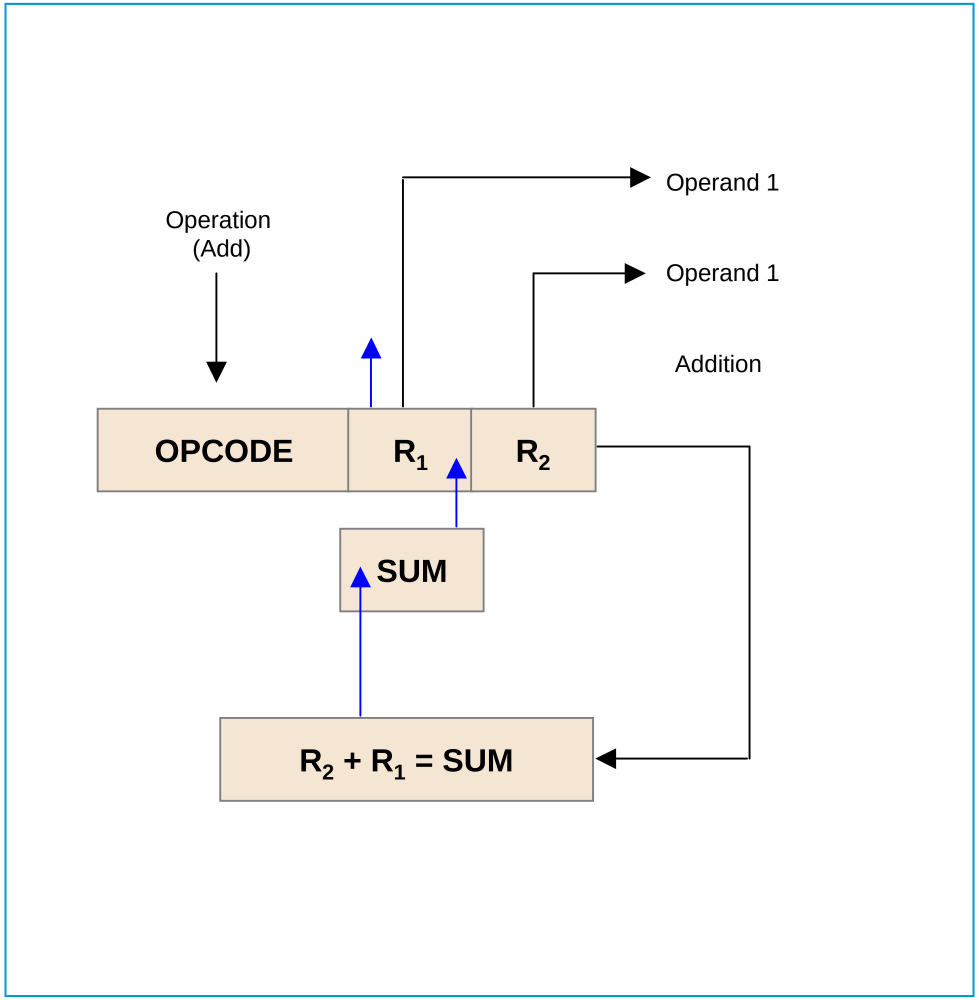
In other cases, the length of a storage operand is explicitly specified as part of the instruction (L).

Continued…

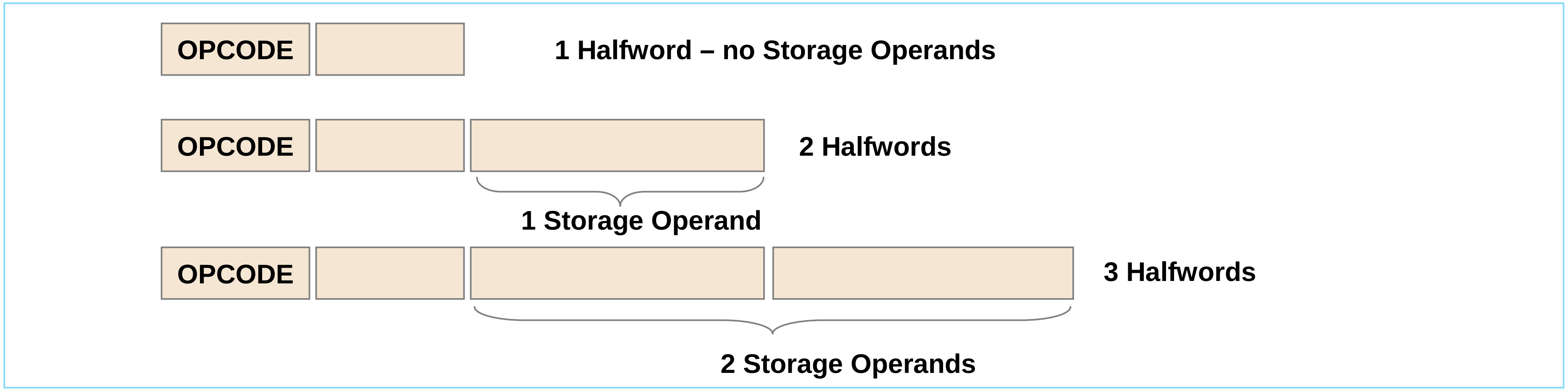
Types of Operands (cont'd)

Most instructions have two operands. In most instructions, the result of the operations replaces the first operand. For example, Add has two operands.

The Add instruction (A) fetches both operands, adds them together and replaces the first operand with the sum.



Instruction Lengths



What are the different instruction lengths?

There are 3 possible instruction lengths, 1, 2 or 3 halfwords. They are:

- 1 halfword instructions have no storage operands
- 2 halfword instructions have one storage operand
- 3 halfword instructions have 2 storage operands

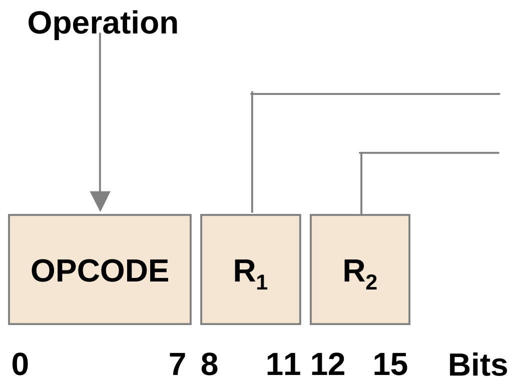
Unit: S/390 Addressing and Instructions Topic: Machine Instruction Formats

RR Instruction

Format

The instruction is 16 bits or one halfword in length. The first byte contains the opcode, and the second byte contains the two operands, both of which are registers.

RR Instruction:



Note! In numbering the bits in the instruction format, start from zero, not one.

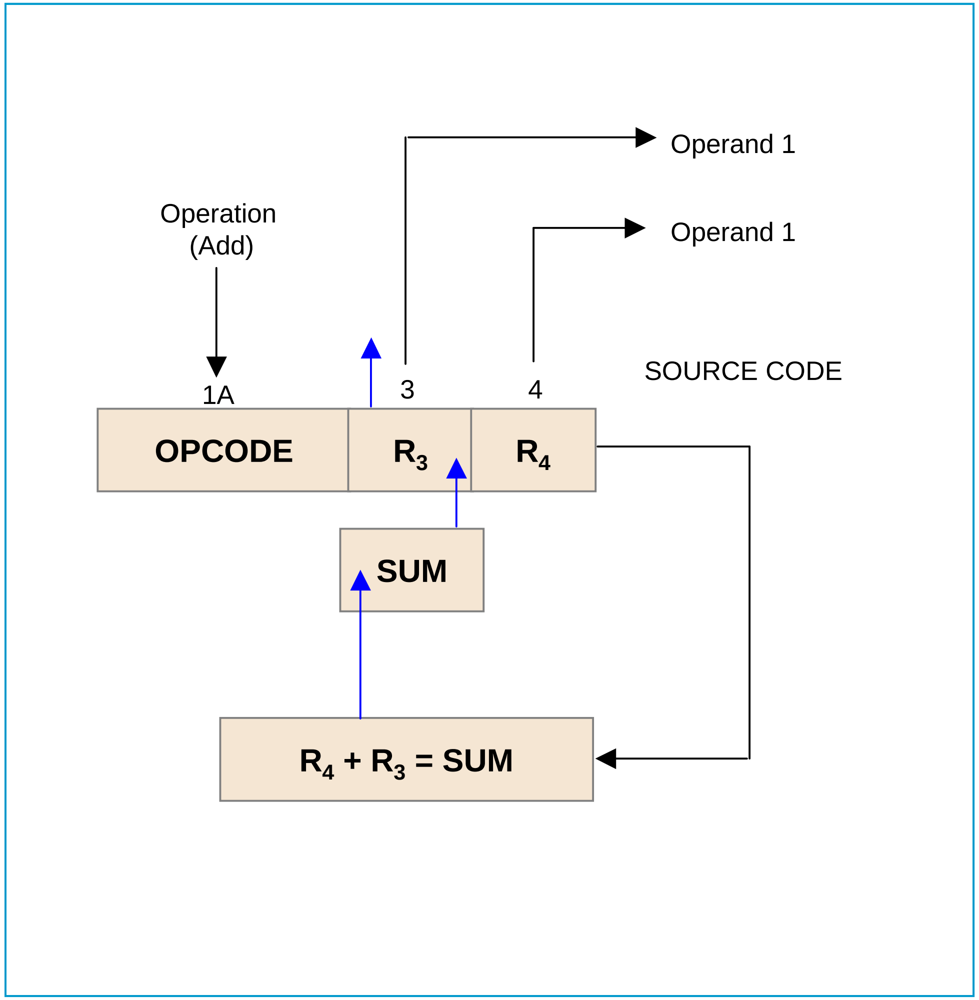
Continued…

RR Instruction (cont'd)

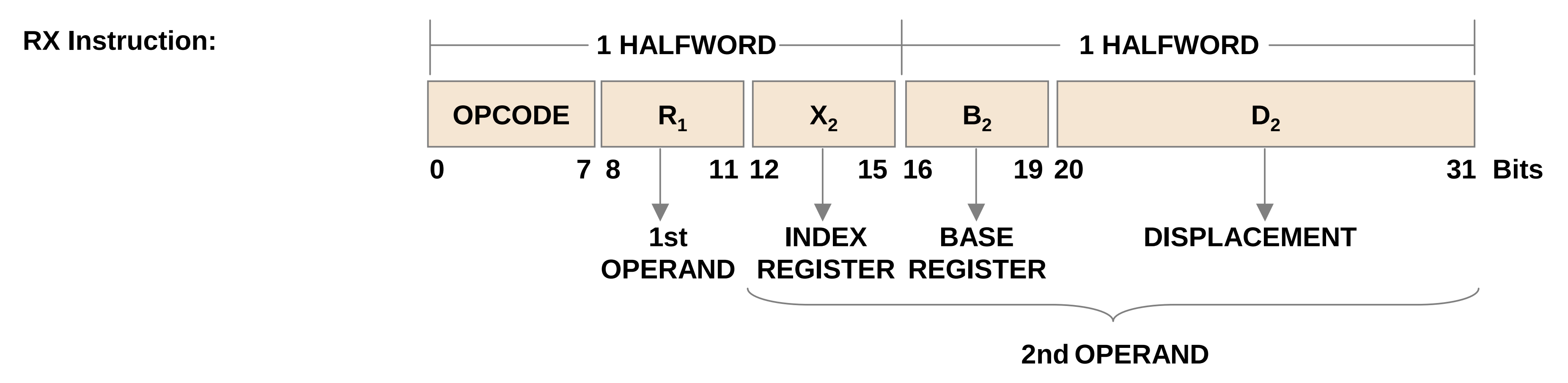
An example of RR Instruction is:
1A34

1A is the operation code for the Add Register instruction. The first operand is R3. The second operand is R4.

When this instruction is executed, the contents of R4 would be added to the contents of R3 and the sum would be placed in R3, replacing the first operand.



RX Instruction



Format

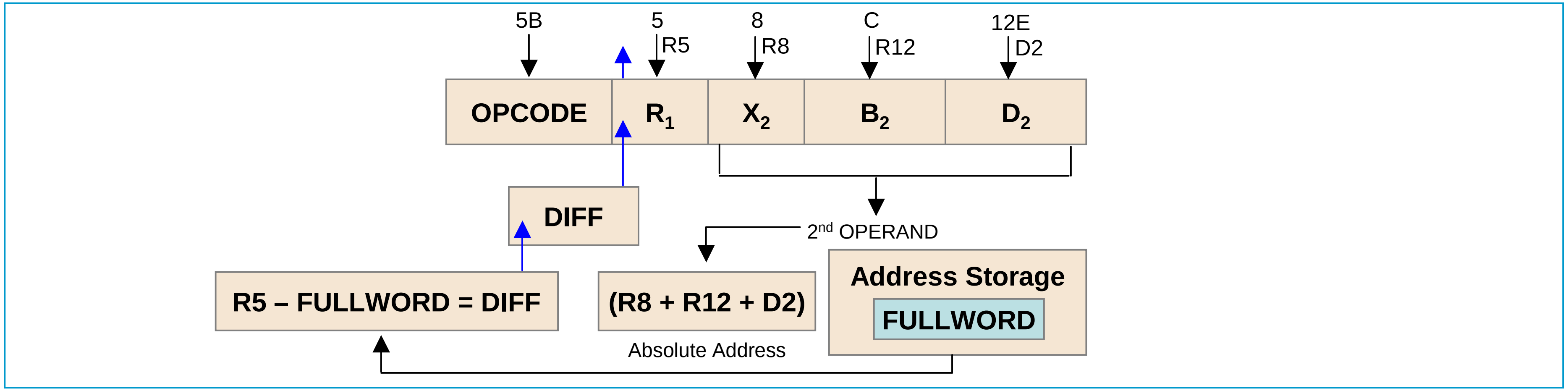
This instruction is 32 bits (or 2 halfwords) in length. The first byte contains the opcode. The following nibble contains the first operand, which is a register.

The remaining fields in the instruction contain the three components that specify the second operand. They are:

- X2, the index register specification
- B2, the base register specification
- D2, the 12 bit displacement of the second operand

Continued…

RX Instruction (cont'd)



An example of RX instruction is:

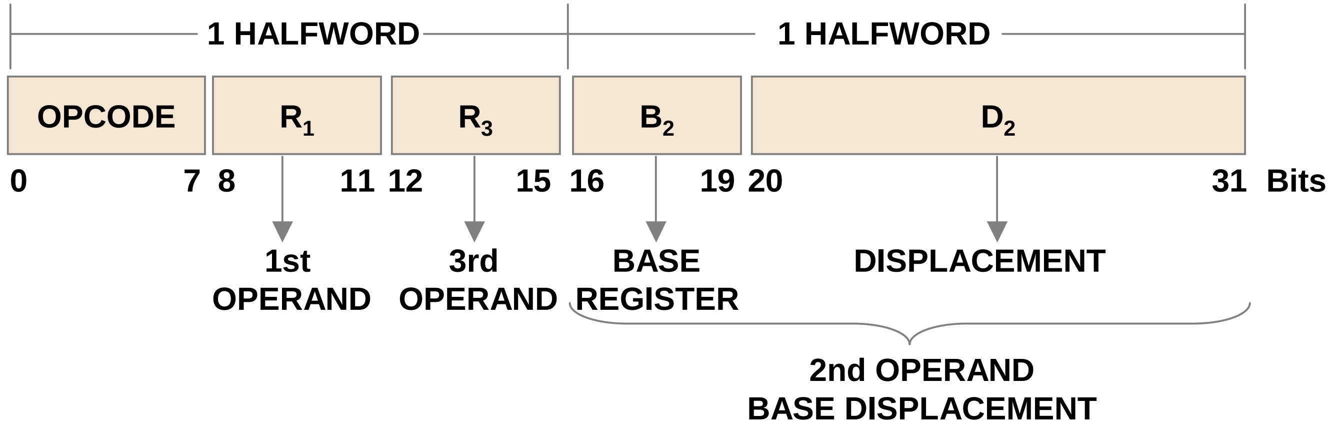
5B58C12E

5B is the operation code for the Subtract instruction. The first operand is R5. The second operand is the fullword at the storage location whose absolute address is formed by adding together the contents of R8, the contents of R12, and the value of the displacement, 12E.

When this instruction is executed, the fullword from storage will be subtracted from the value in R5 and the difference placed in R5.

RS Instruction

RS Instruction:



1 HALFWORD 1 HALFWORD

OPCODE R_1 R_3 B_2 D_2

0 7 8 11 12 15 16 19 20 31 Bits

1st OPERAND

3rd OPERAND

BASE REGISTER

DISPLACEMENT

2nd OPERAND BASE DISPLACEMENT

Format

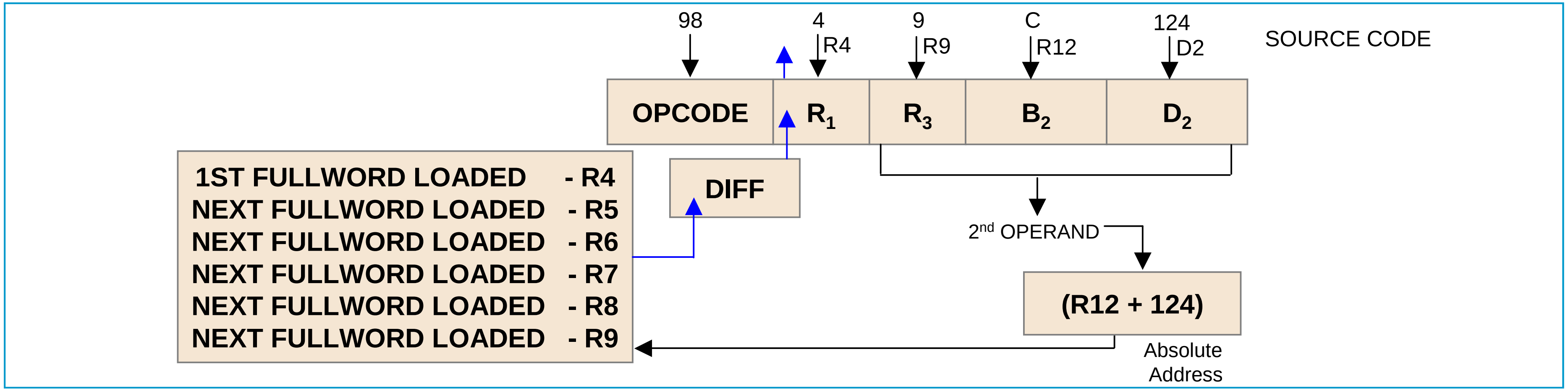
The instruction is 32 bits or 2 halfwords in length. The first byte of the instruction is the opcode. This is followed by the first and third operands, both registers. Finally the second operand, the storage operand, specified in base displacement form.

Note!

Note that this instruction has three operands not just two.

Continued…

RS Instruction (cont'd)


98
4
R4
9
R9
C
R12
124
D2
SOURCE CODE
OPCODE
R_1
R_3
B_2
D_2
1ST FULLWORD LOADED - R4
NEXT FULLWORD LOADED - R5
NEXT FULLWORD LOADED - R6
NEXT FULLWORD LOADED - R7
NEXT FULLWORD LOADED - R8
NEXT FULLWORD LOADED - R9
DIFF
2nd OPERAND
(R12 + 124)
Absolute Address

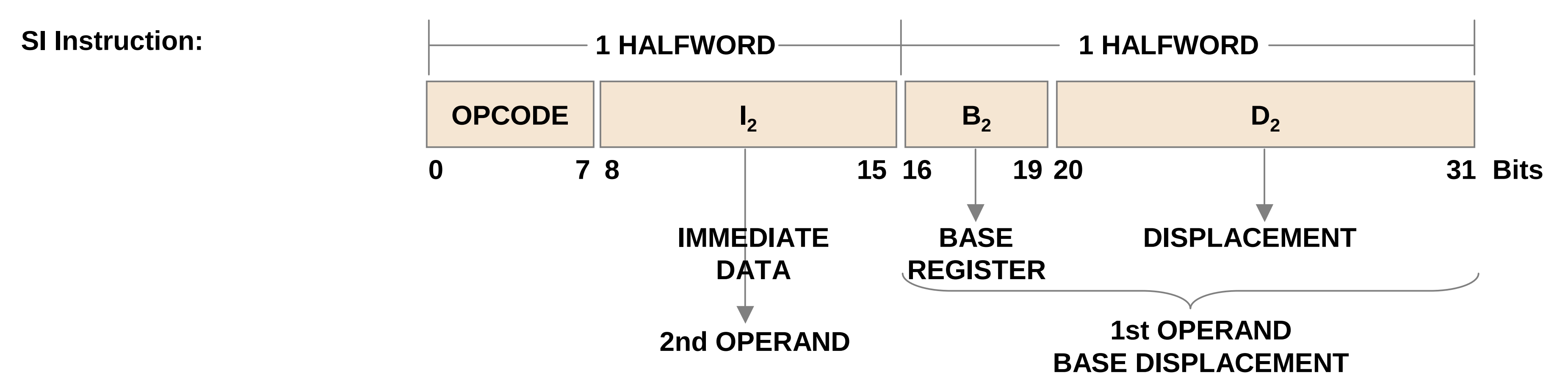
Format

9849C124

98 is the opcode for the Load Multiple instruction. The first operand is R4. The third operand is R9.

The second operand is the 6 consecutive full words starting at the storage location, whose absolute address is formed by adding the displacement 124 to the contents of R12, the base register. When this instruction is executed, the first fullword at the storage location would be loaded into R4, the next into R5, the next into R6, the next into R7, the next into R8 and the next into R9.

SI Instruction

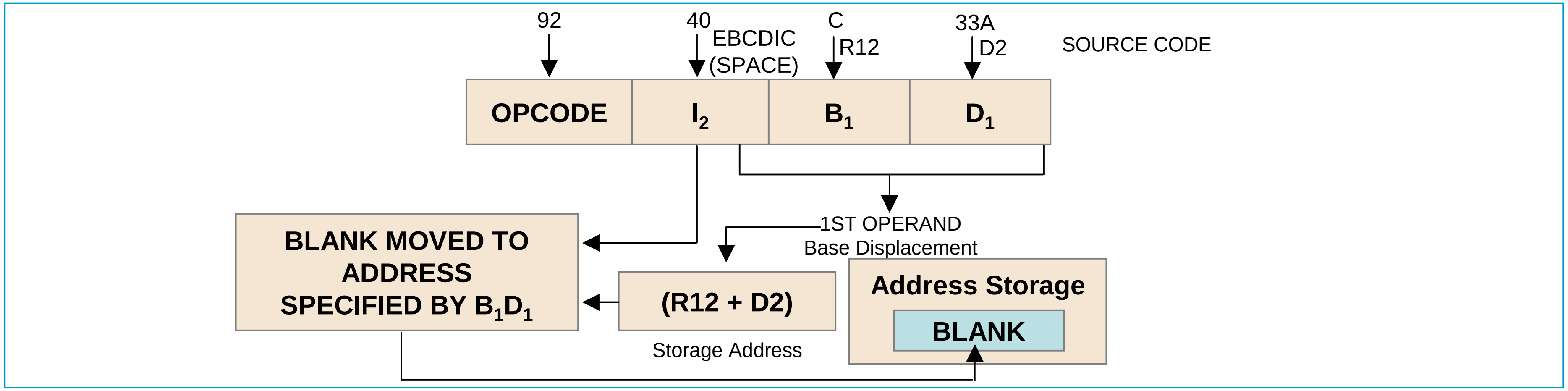


Format

The instruction is 32 bits or two halfwords in length. The first byte of the instruction is the opcode. The next byte is the second operand, an immediate storage operand. This byte contains the actual operand, not an address. The operand is exactly one byte long. The rest of the instruction is the base and displacement specifying the first operand.

Continued…

SI Instruction (cont'd)



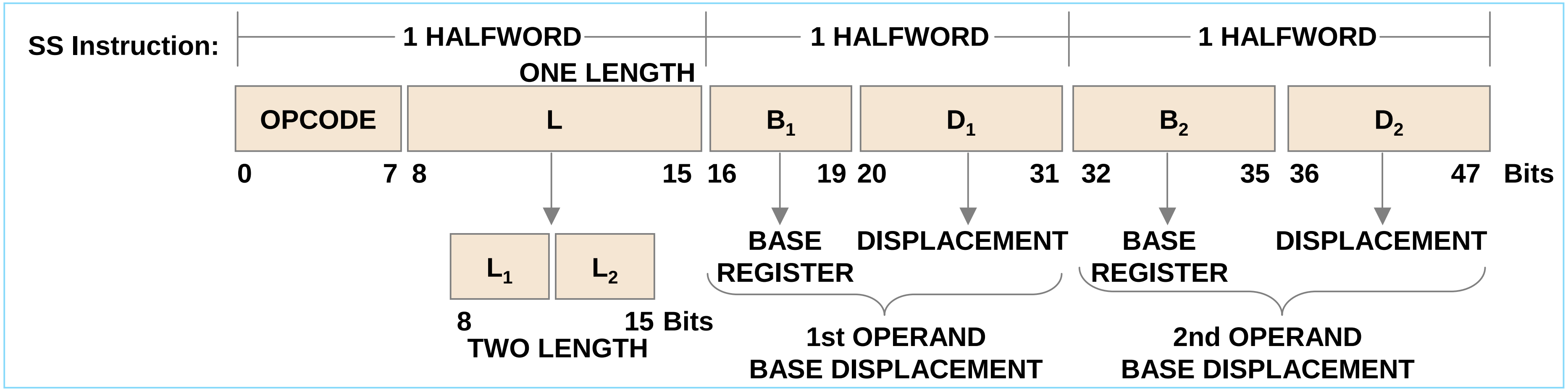
An example of SI instruction is:

9240C33A

92 is the opcode for the Move Immediate instruction. 40, the EBCDIC code for space, is the second operand.

The first operand is the byte at the absolute storage address formed by adding the displacement 33A to the contents of R12, the base register. When this instruction is executed a blank would be moved to the address specified by the first operand.

SS Instruction

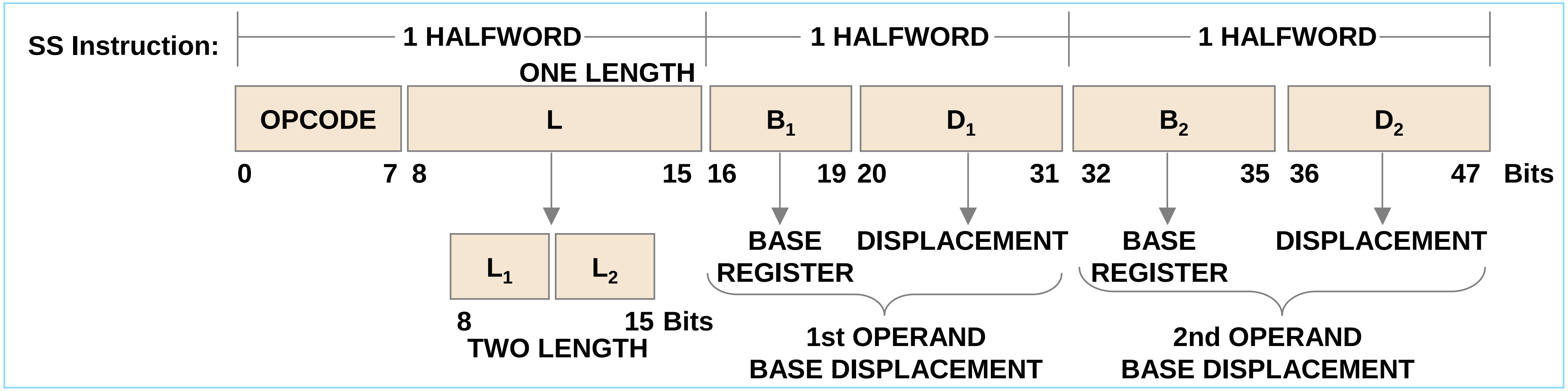


What are the two formats of SS instructions?

The first format has single length specification, which applies to both of its operands. The second has a separate length specified for each operand.

Continued…

SS Instruction (cont'd)



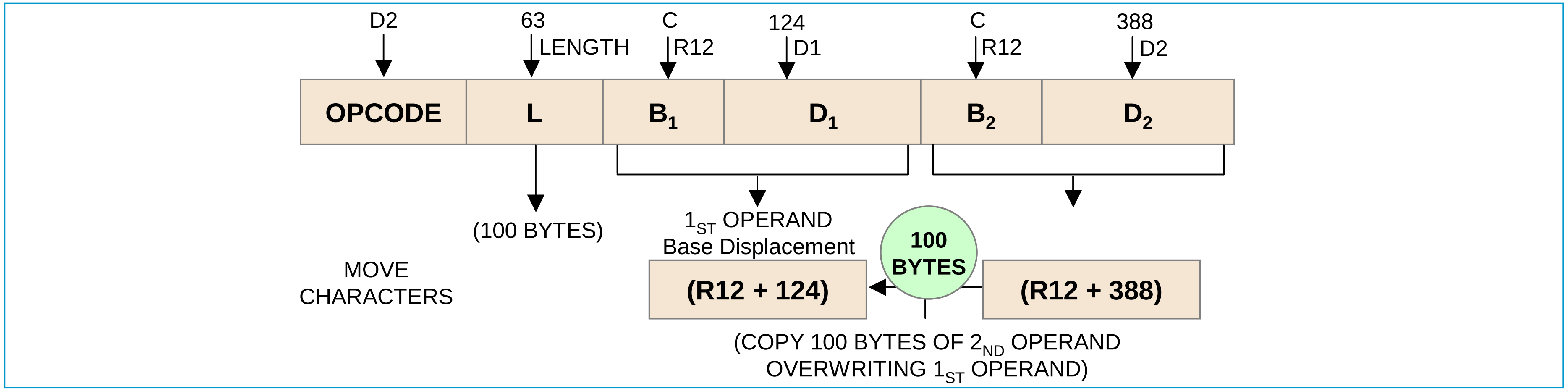
When there are two lengths, each is specified as a nibble, and can contain values from 0 – 15. when there is a single length specification it is a byte in length, and can contain values from 0 – 255.

All lengths in machine instructions are coded as 1 less than the actual length. This is because a zero length dosen't make sense for an operation, and so in order to provide maximum facility, a machine length specification of 0, represents a real length of 1, 1 represents 2, and so on.

The two SS formats are shown above. The instructions are 48 bits or 3 halfwords in length. The first byte is the opcode. The next byte is the length or lengths. This is followed by operand one and operand two, both specified in base displacement format.

Continued…

SS Instruction (cont'd)



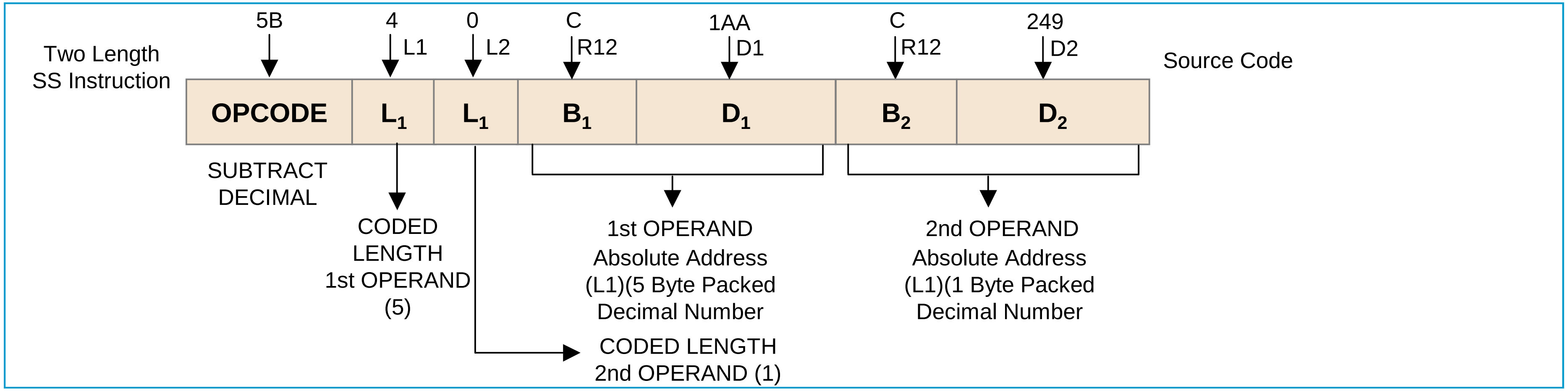
An example of one length SS instruction is:

D263C124C388

D2 is the opcode for the Move Characters instruction. 63 is the machine coded length, equivalent to an actual length of 100 bytes.

The first operand is the 100 character string whose absolute address is formed by adding the displacement of 124 to the contents of R12. The second operand is the 100 character string whose absolute address is formed by adding the displacement of 388 to the contents of R12. When this instruction is executed, the 100 bytes of the second operand, would be copied to the first operand, destroying the original value of the first operand.

SS Instruction – An Example


Two Length
SS Instruction
5B
4
L1
0
L2
C
R12
1AA
D1
C
R12
249
D2
Source Code
OPCODE
L_1
L_1
B_1
D_1
B_2
D_2
SUBTRACT DECIMAL
CODED LENGTH 1st OPERAND (5)
1st OPERAND Absolute Address (L1)(5 Byte Packed Decimal Number
2nd OPERAND Absolute Address (L1)(1 Byte Packed Decimal Number
CODED LENGTH 2nd OPERAND (1)

An example of two length SS instruction is:
5B40C1AAC249

5B is the opcode for the Subtract Decimal instruction. 4 is the machine coded length of the first operand, equivalent to an actual length of 5. 0 is the machine coded length of the second operand, equivalent to an actual length of one.

Continued…

SS Instruction – An Example (cont'd)

An example of two length SS instruction (cont'd)

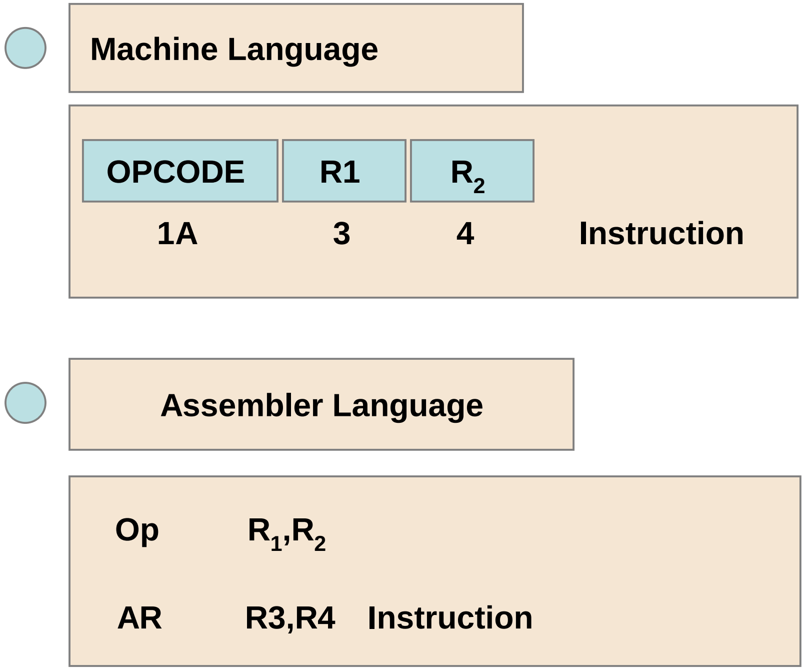
The first operand is the 5 byte packed decimal number whose absolute address is formed by adding the displacement of 1AA to the contents of R12. The second operand is the 1 byte packed decimal number whose absolute address is formed by adding the displacement of 249 to the contents of R12. When this instruction is executed the value of the second operand will be subtracted from the value of the first, and the difference would replace the first operand.

Difference in RR Format

In the RR format, the components for both Machine language and Assembler language follow in the same order; operation, operand 1 and operand 2.

RR Format

Example Code: 1A34



Machine Language

OPCODE	R1	R_2	
1A	3	4	Instruction

Assembler Language

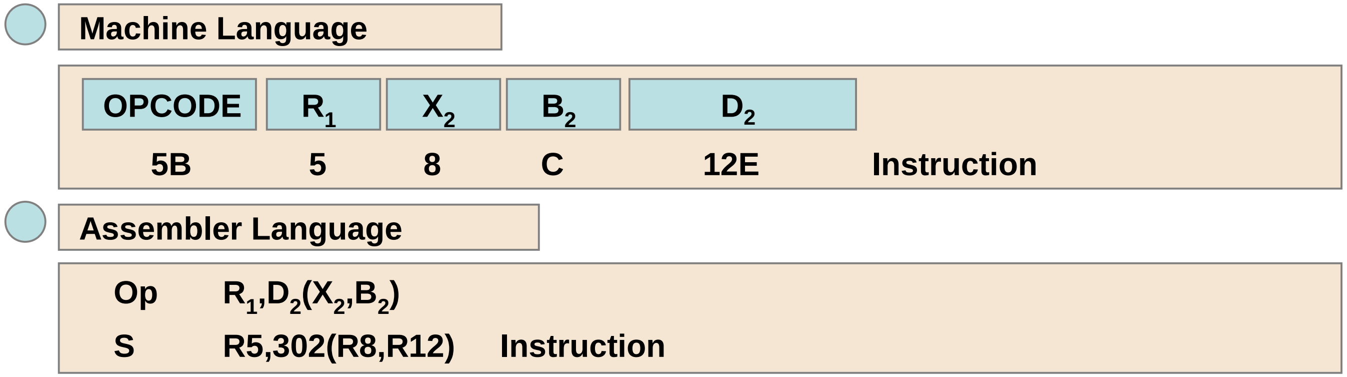
Op	R_1,R_2	
AR	R3,R4	Instruction

Difference in RX Format

RX Format

Example Code:

5B58C12E



Machine Language

OPCODE	R_1	X_2	B_2	D_2	
5B	5	8	C	12E	Instruction

Assembler Language

Op	$R_1,D_2(X_2,B_2)$	
S	R5,302(R8,R12)	Instruction

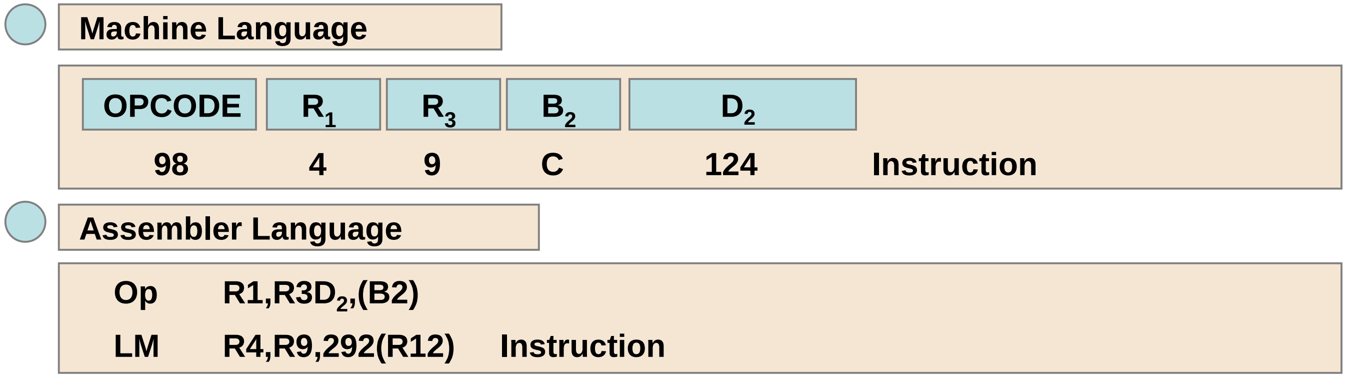
With RX format where we have storage operands, the syntax is quite different. Start with operation, followed by operand1. Operand2 is specified with the displacement first. Notice displacement is decimal in the Assembler language and hexadecimal in machine language.

This is followed by the index register and the base register, separated by a comma, and in parentheses (example: R8, R12).

Continued…

Difference in RX Format (cont'd)

Example Code:
9849C124



Machine Language

OPCODE	R_1	R_3	B_2	D_2	
98	4	9	C	124	Instruction

Assembler Language

```
Op      R1,R3D2,(B2)
LM      R4,R9,292(R12)    Instruction
```

With RS format, again notice the opcode and register operands in Assembler Language parallel the machine language format. The storage operand is specified as displacement (decimal), and is followed by the base register in parentheses.

Difference in SI Format

SI Format,
Example Code:
9240C33A

Machine Language

OPCODE	I_2	B_1	D_1	
98	4	9	33A	Instruction

Assembler Language

```
Op     D1,(B1),I2
MVI    826(R12),X'40'     Instruction
```

In SI format, the order of the operands is different in the two formats. In Machine Language, the immediate operand I_2, precedes the storage operand. In Assembler Language, the storage operand comes first. The immediate operand is specified as a hex value of 40 in the notation shown.

Difference in SS1 Format

Example Code:
D263C124C388

Machine Language

OPCODE	L	B_1	D_1	B_2	D_2	
D2	63	C	124	C	388	Instruction

Assembler Language

Op	D1,(L,B1),D_2(B_2)	
MVC	292(100,R12),904(R12)	Instruction

When a length is specified with a storage operand, it is specified as the actual length, and precedes the base register, followed by a comma, within parentheses, for example (100,R12).

If there is only one length, it is placed with the first operand, not the second. Note in the example that the length is 100 decimal. $100_{10} = 64_{16}$. The machine length is one less than the actual length, so the machine length is 63.

Difference in SS2 Format

SS2 Format
Example Code:
5B40C1AAC249

Machine Language

OPCODE	L_1	L_2	B_1	D_1	B_2	D_2	
5B	4	0	C	1AA	C	249	Instruction

Assembler Language

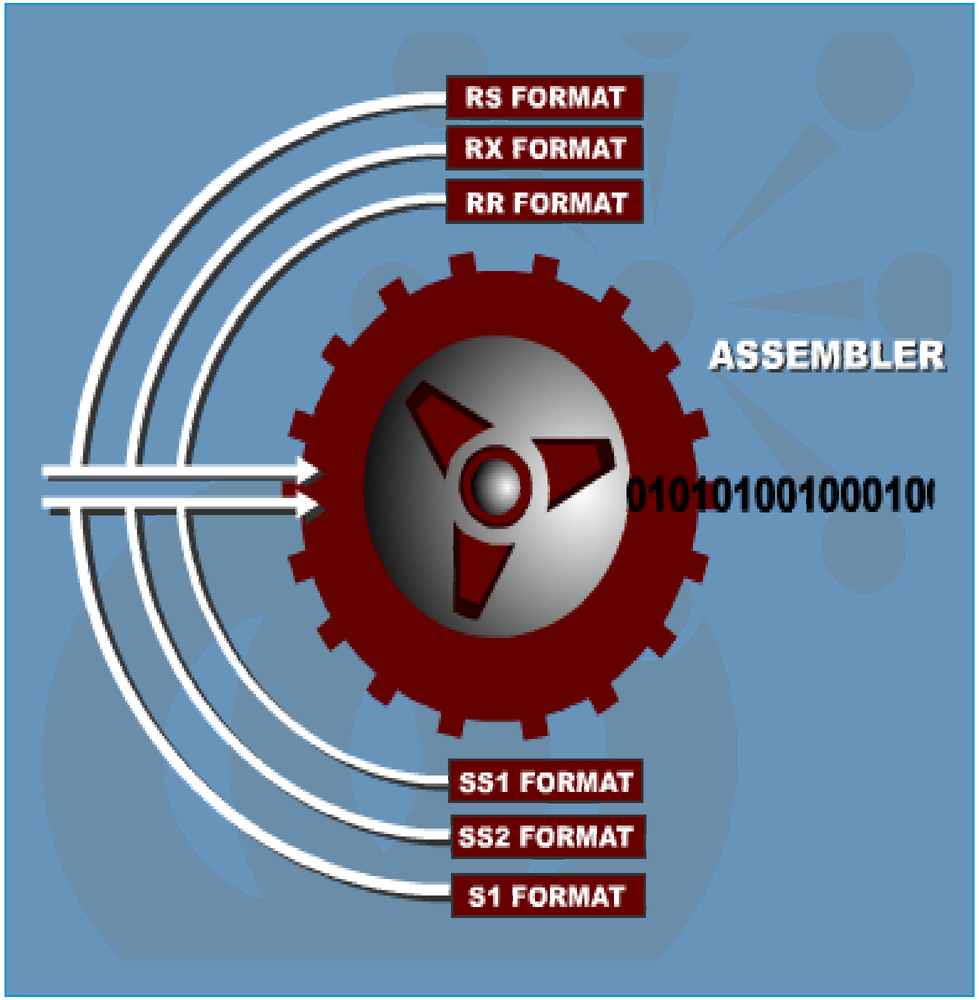
Op	D1,(L_1,B_1),D2(L2,B2)	
SP	426(5,R12),585(1,R12)	Instruction

This example follows the same pattern as the previous one. In this case there are two lengths, one for each operand. Lengths are coded before the base register, within parentheses.

Assembler Program

What is an Assembler Program?

Looking at all the instruction formats for both Assembler and Machine Languages may be somewhat intimidating. However the job of the Assembler programmer is not as complex as you might expect. The Assembler program converts your Assembler statements to Machine Language for you.



Continued…

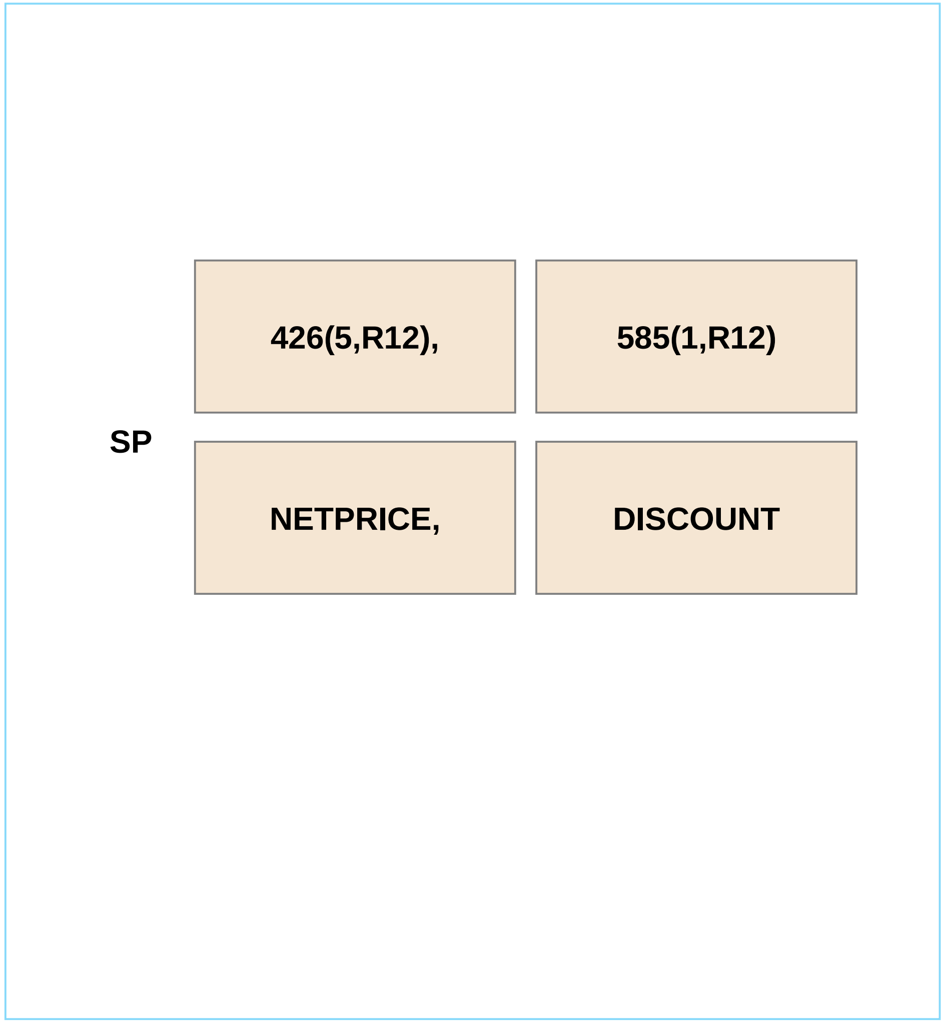
Assembler Program (cont’d)

Thus, instead of writing an instruction like this:

```
SP 426(5,R12),585(1,R12)
```

You would probably write something like this:

```
SP NETPRICE, DISCOUNT
```



SP

426(5,R12), 585(1,R12)

NETPRICE, DISCOUNT

Assembler Language Statements

What are Statements?

Assembler Language programs are made up of statements. Statements are composed of characters. Lower case alphabetic characters are considered equivalent uppercase characters by the Assembler, except when used in quoted strings.

Characters:

Alphabetic characters	**a –z and A- Z**
National characters	**@, $, and #**
Underscore	**_**
Digits	**0 – 9**
Special characters	**+ - , = , * ( ) ‘ / &** **blank**

Continued…

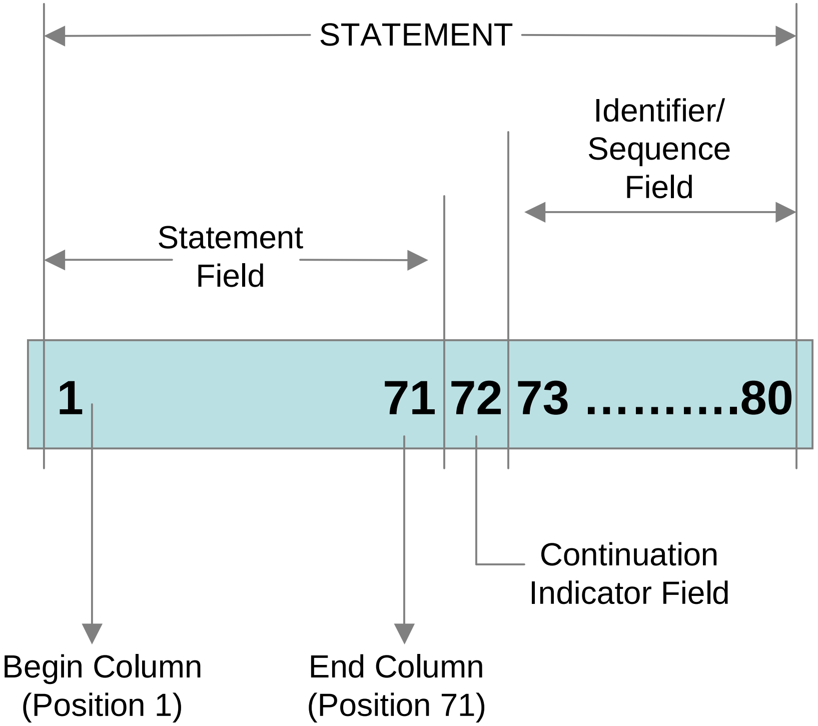
Assembler Language Statements (cont'd)

What constitutes Assembler Language Statements?

Assembler Language Statements are coded in 80 character records or lines. By default, each 80 character line consists of 3 fields; positions 1-71 constitute the statement field, 72 is the continuation indicator field, and 73-80 is the identifier/sequence field.

Assembler Language statements and comments must be written in the statement field. It consists of:

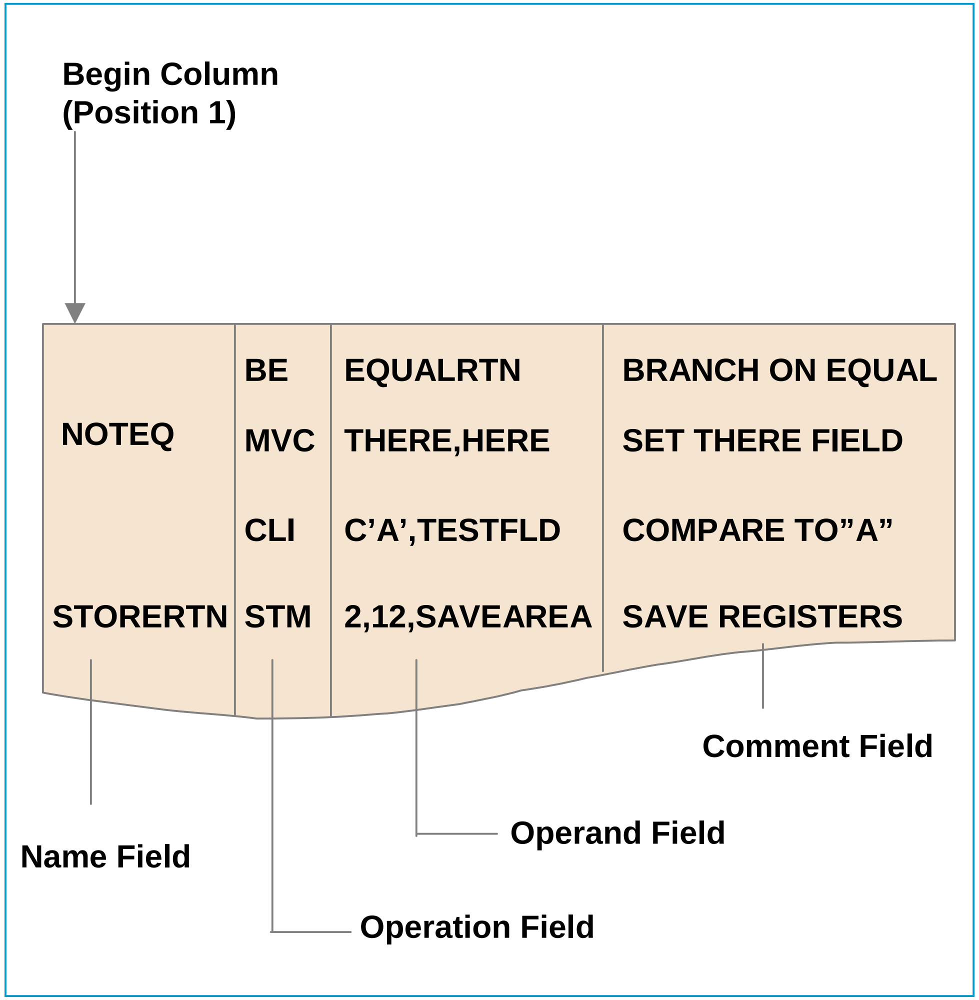
- Begin Column - The first position of the statement field
- Continue Column - second and subsequent statements of a continued statement begin
- Continuation Indicator Field - The position following the end column


STATEMENT
Identifier/ Sequence Field
Statement Field
1
71
72
73 ……….80
Continuation Indicator Field
Begin Column (Position 1)
End Column (Position 71)

Assembler Language Instruction Statements

Assembler Language Instruction Statement consists of up to 4 fields, specified in the following order:

1. Name
2. Operation
3. Operand
4. Comments



```
         BE    EQUALRTN         BRANCH ON EQUAL
NOTEQ    MVC   THERE,HERE       SET THERE FIELD
         CLI   C’A’,TESTFLD     COMPARE TO”A”
STORERTN STM   2,12,SAVEAREA    SAVE REGISTERS
```

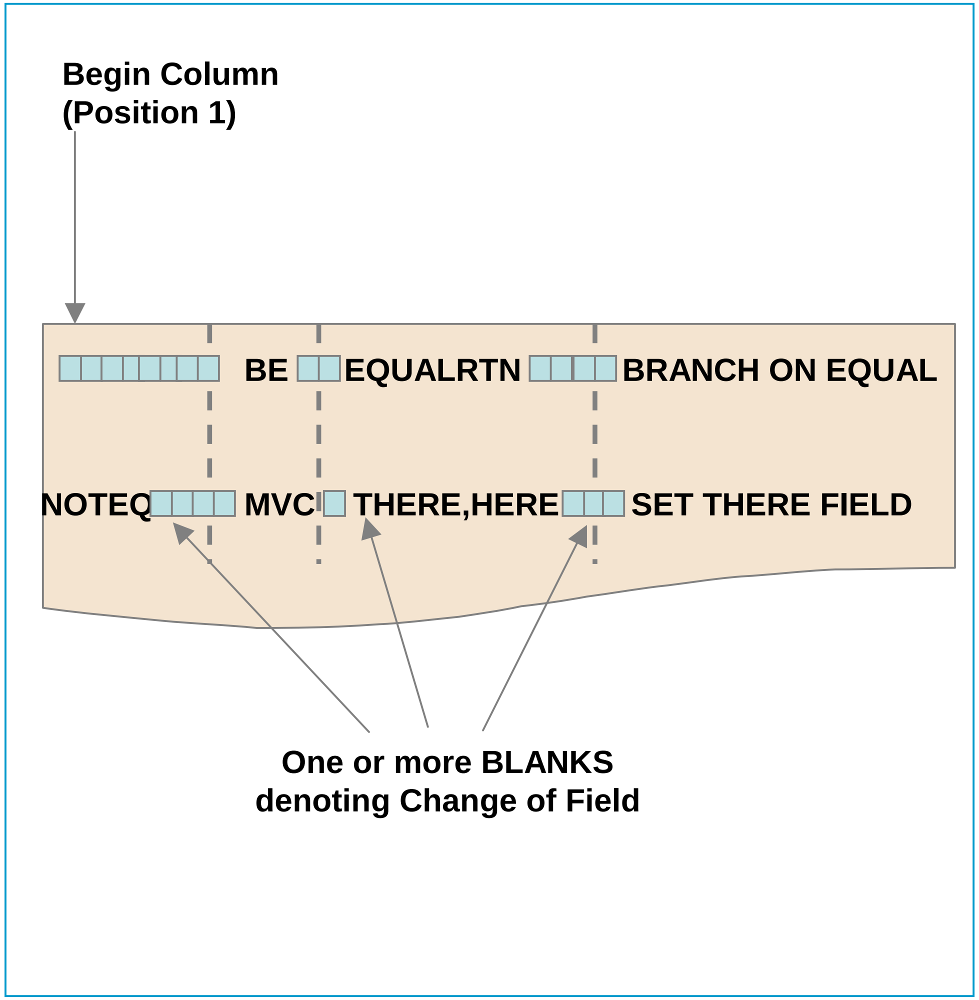
Continued…

Assembler Language Instruction Statements (cont'd)

Assembler Language instructions are free format, with one or more blanks delimiting fields. The name field and operation field is separated by one or more blanks, as are the operation and the operand, and the operand and the comment.

Embedded blanks are not allowed in the operand field, unless they are part of a quoted string. Otherwise they would indicate the end of the operand field.

The name field, if present, must begin in the begin column. If the name field is omitted, the operation field must start after the begin column.



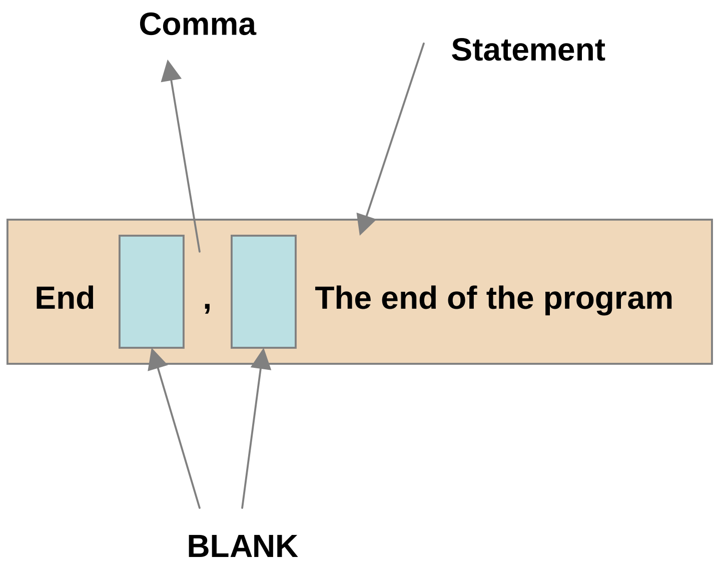
Continued…

Assembler Language Instruction Statements (cont'd)

Although the Assembler allows free format statements, most programmers align their fields (operation, operand and comments) in specific positions for readability.

If you wish to code a comment field on a statement with no operand field, you must indicate to the Assembler, that the operand field is missing. You do this by coding a comma, preceded and followed by a blank, between the operation and the comment. An example of this is shown on the right.

Coding a Comment Field with no Operand Field:



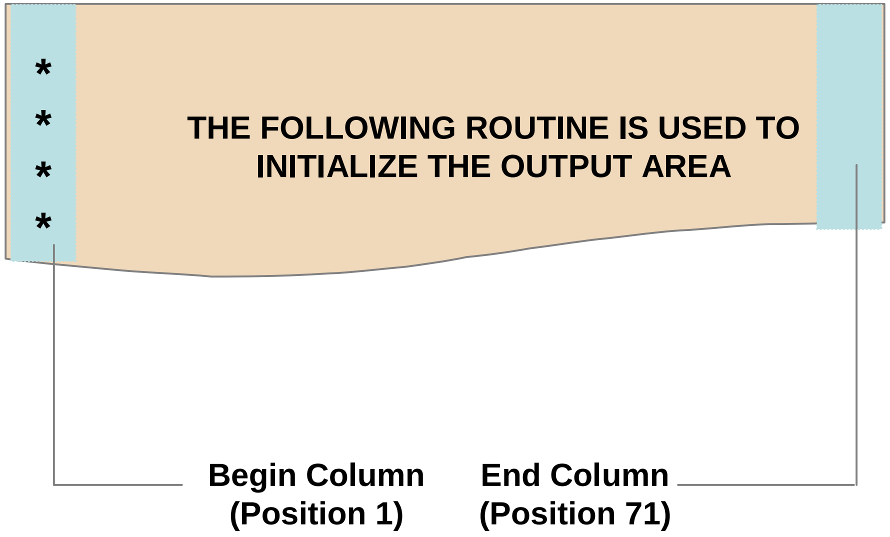
Use of Comments

How can you use comments effectively?

As noted previously comments can be included as the fourth field on any Assembler Language instruction statement.

It is also possible to have statements in your program consisting entirely of comments.

An Assembler Language comment statement has an asterisk in the statement in the begin column. The rest of the statement, up to and including the end column, is interpreted as a comment.


*
*
*
*
THE FOLLOWING ROUTINE IS USED TO INITIALIZE THE OUTPUT AREA
Begin Column (Position 1)
End Column (Position 71)

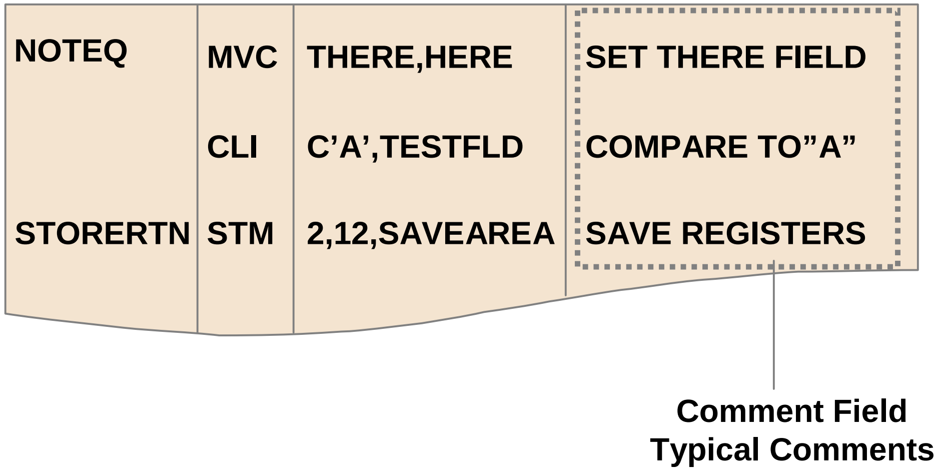
Use of Comments (cont'd)

How can you use comments effectively?

It is good programming practice to include plenty of comments. It is desirable to clearly document what your code is doing when using a low level language like Assembler Language.

A block of comment statements should describe the purpose of each module and of each major section of code.

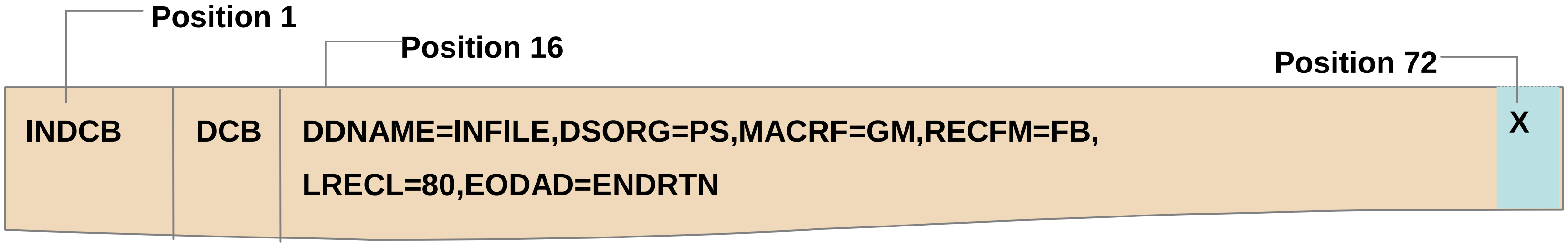
Comments should be coded on instruction statements to indicate the purpose of each individual statement.



```
NOTEQ     MVC  THERE,HERE      SET THERE FIELD
          CLI  C'A',TESTFLD    COMPARE TO"A"
STORERTN  STM  2,12,SAVEAREA   SAVE REGISTERS
```

Comment Field
Typical Comments

Formatting Assembler Language Statements



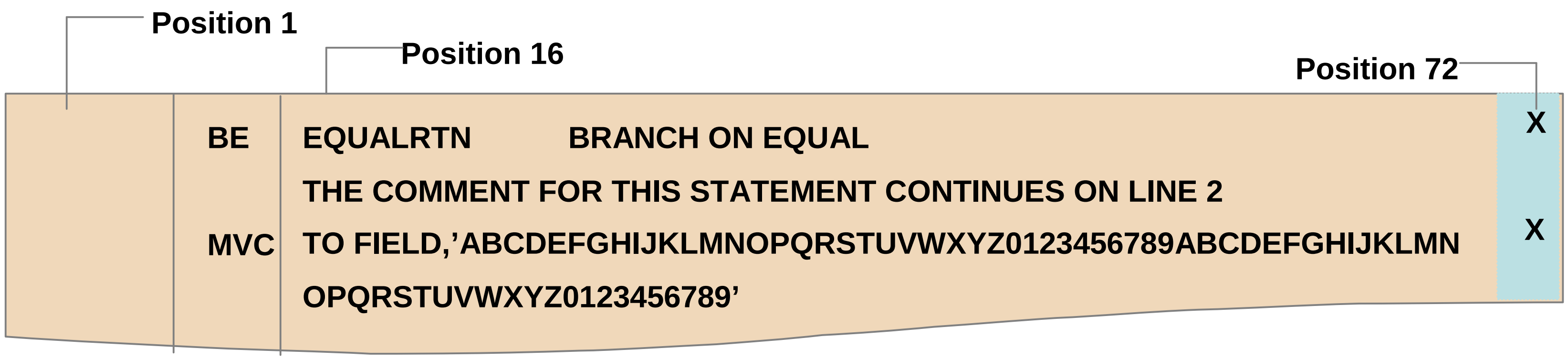
```
INDCB    DCB    DDNAME=INFILE,DSORG=PS,MACRF=GM,RECFM=FB,         X
                LRECL=80,EODAD=ENDRTN
```

Most Assembler Language statements will easily fit in one line. Sometimes, particularly with complex macro instructions, a statement will not fit on a single line, between the start and end columns.

Note!

The example above shows a statement continued on the next line.

Breaking Statements



```
         BE    EQUALRTN      BRANCH ON EQUAL                          X
               THE COMMENT FOR THIS STATEMENT CONTINUES ON LINE 2
         MVC   TO FIELD,'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ABCDEFGHIJKLMN X
               OPQRSTUVWXYZ0123456789'
```

Following are the two ways of breaking a statement for continuation:

1. Code the statement up to the end column, put a non-blank character in the continuation indicator column and resume coding in the continuation column of the next line

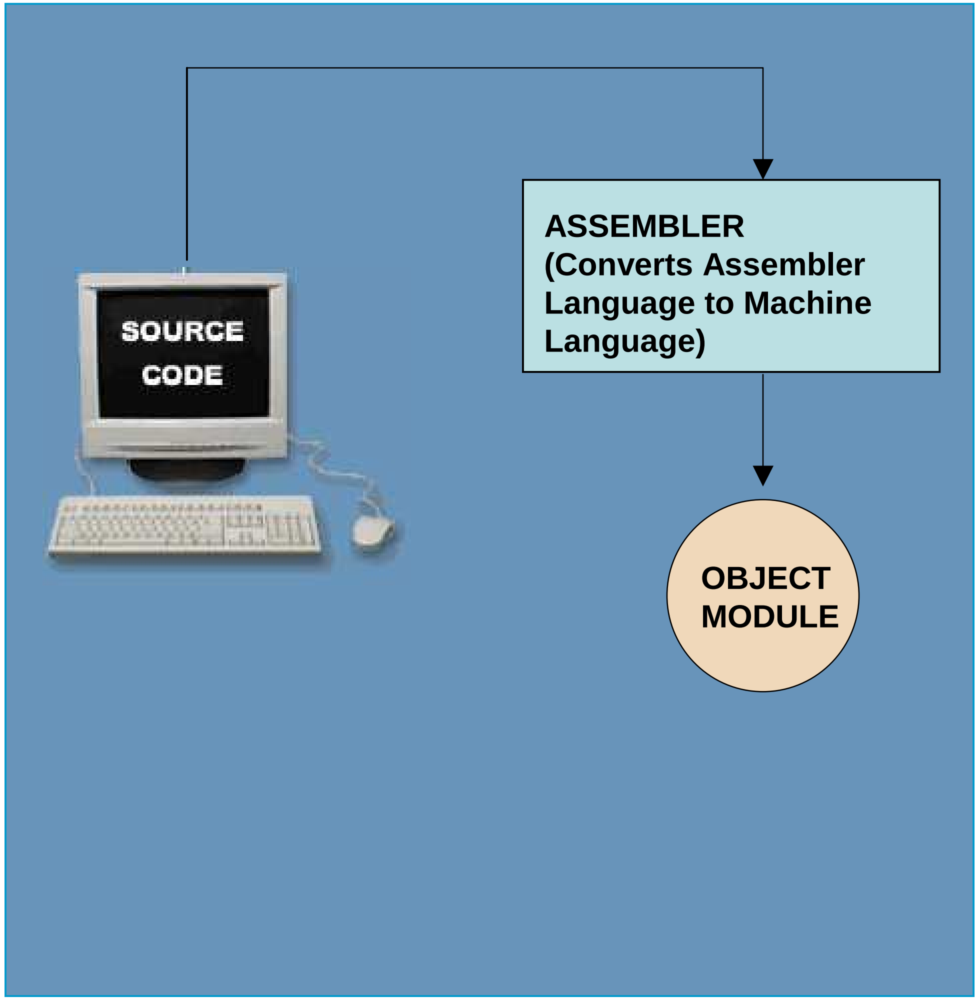
2. Code the statement to the end of a complete operand and its following comma, put a non-blank character in the continuation indicator column and resume coding with the start of the next operand in the continuation column of the next line

Assembler Functions

What are the functions of an Assembler?

The major purpose of the Assembler is to take a source program, written in Assembler Language and convert it to an object module in Machine Language. This conversion includes:

- Converting Assembler Language mnemonic opcodes into Machine Language numeric opcodes
- Converting the Assembler Language instruction formats into Machine Language forms
- Converting the numeric notation used in the source program (decimal by default) to hexadecimal form used in the Machine Language
- Determining the address of all the instructions and data areas used in the program
- Converting reference symbols into base displacement addresses



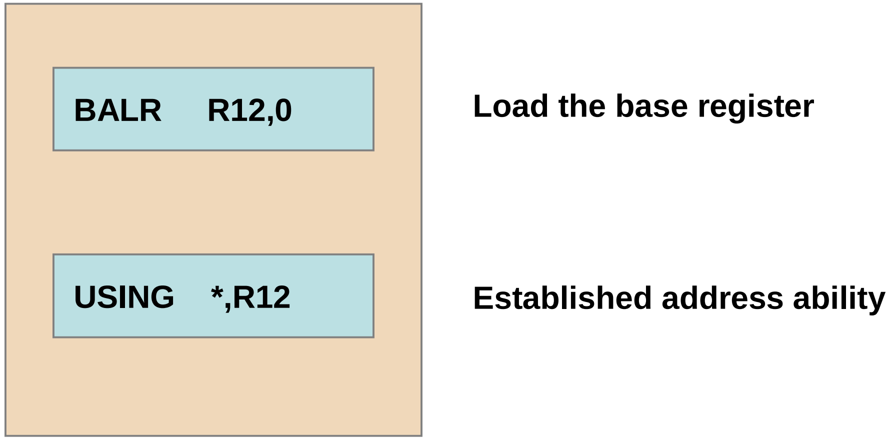
Converting Addresses to Base-Displacement Format

What is USING Instruction?

In order for the Assembler to convert addresses to base-displacement format, you must tell the Assembler what register to use as a base register, and what value the register will contain at execution time. This information is provided with the USING instruction.

You must also include machine instructions in your program, to load the proper value into the base register at execution time. A standard way of accomplishing these two actions is with the following instructions, included as part of the program initiation:

BALR R12,R0 and USING*, R12



```
BALR     R12,0
USING    *,R12
```

Load the base register

Established address ability

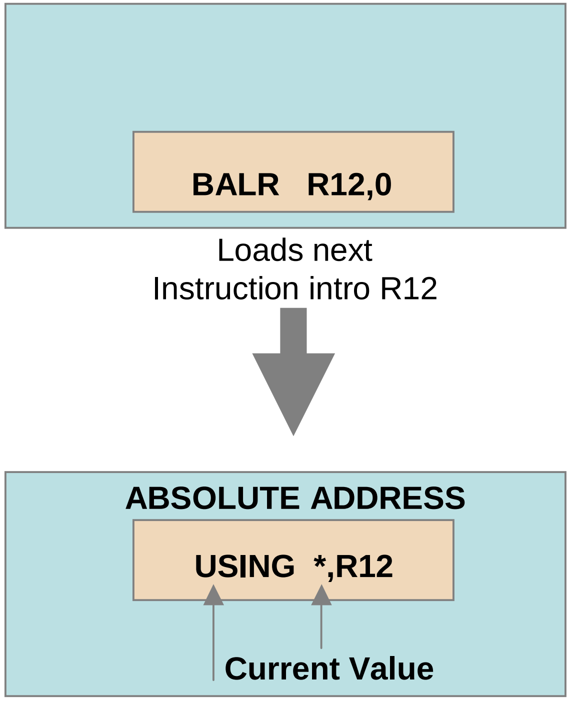
Continued…

Converting Addresses to Base-Displacement Format (cont'd)

The first of these instructions, a Machine instruction, places the absolute address of the next instruction after BALR into register 12. The * symbol is used to represent the current value of the location counter in an Assembler program. The USING instruction tells the Assembler that R12 is to be used as the base register, and that R12 will contain the address of the current instruction.

Since BALR has in fact loaded R12 with the address of the instruction following BALR (which is now the current instruction) this correctly establishes addressability in the program.

To established Program Address ability:



Established R12 as the Base Register

SS Instruction Example

As previously discussed, storage operands can be represented as either symbols, or in base displacement format. The Assembler allows certain portions of the base displacement, base index displacement, or base length displacement forms to be omitted, and uses default values for the omitted parts.

This process is indicated in the table on the right.

After the Assembler produces an object module it can be linked and executed.

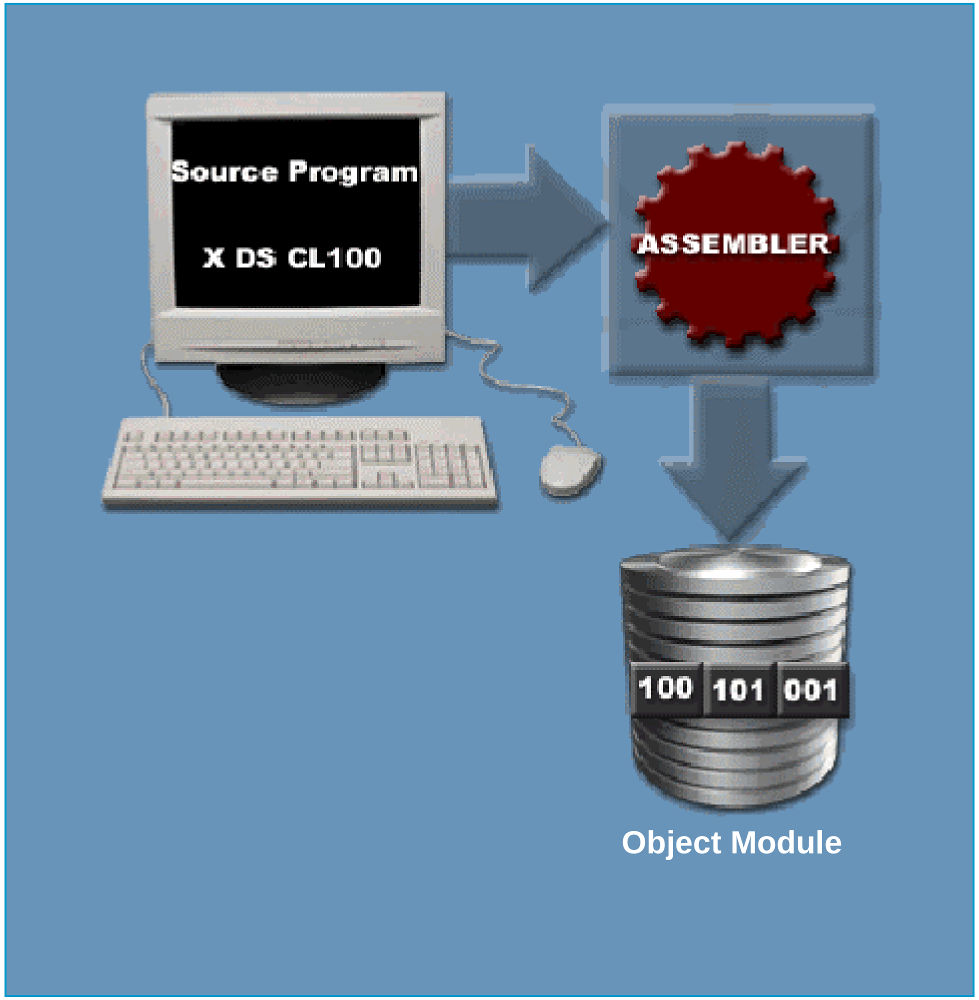
Full Operand	From Omission	Assembler Default
D (B)	D	B=0
D (X,B)	D	X=0 , B=0
	D (X)	B=0
	D (,B)	X=0
D (1,B)	D	B=0 , 1=length Attribute of D
	D (1) D (,B)	B=0 1=length attribute of D

Define Storage Instruction

What is Define Storage (DS) Instruction ?

The Define Storage (DS) Assembler instruction is used to define storage areas within a program.

If you wish to assign a name to the storage being defined, you specify a symbol in the name field of the DS instruction. The operation for defining storage is DS.



Note! Since DS does not define an executable instruction, but rather sets up an area of storage, it is an Assembler instruction.

Continued…

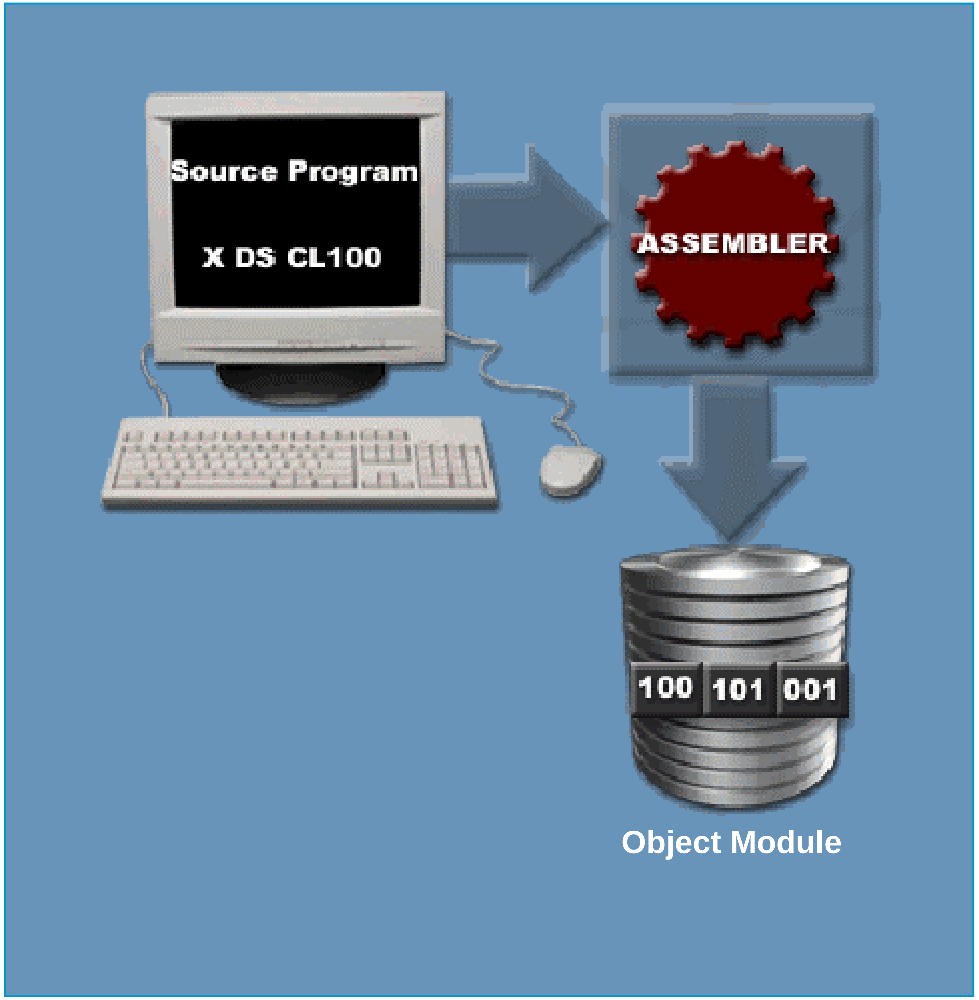
Unit: Defining Data Topic: The Define Storage (DS) Instructions

Define Storage Instruction (cont'd)

What are the components of a DS statement?

The operand field of a DS statement consists of up to 4 components, which are:

- A duplication factor
- A type specifier
- A length modifier
- A nominal value

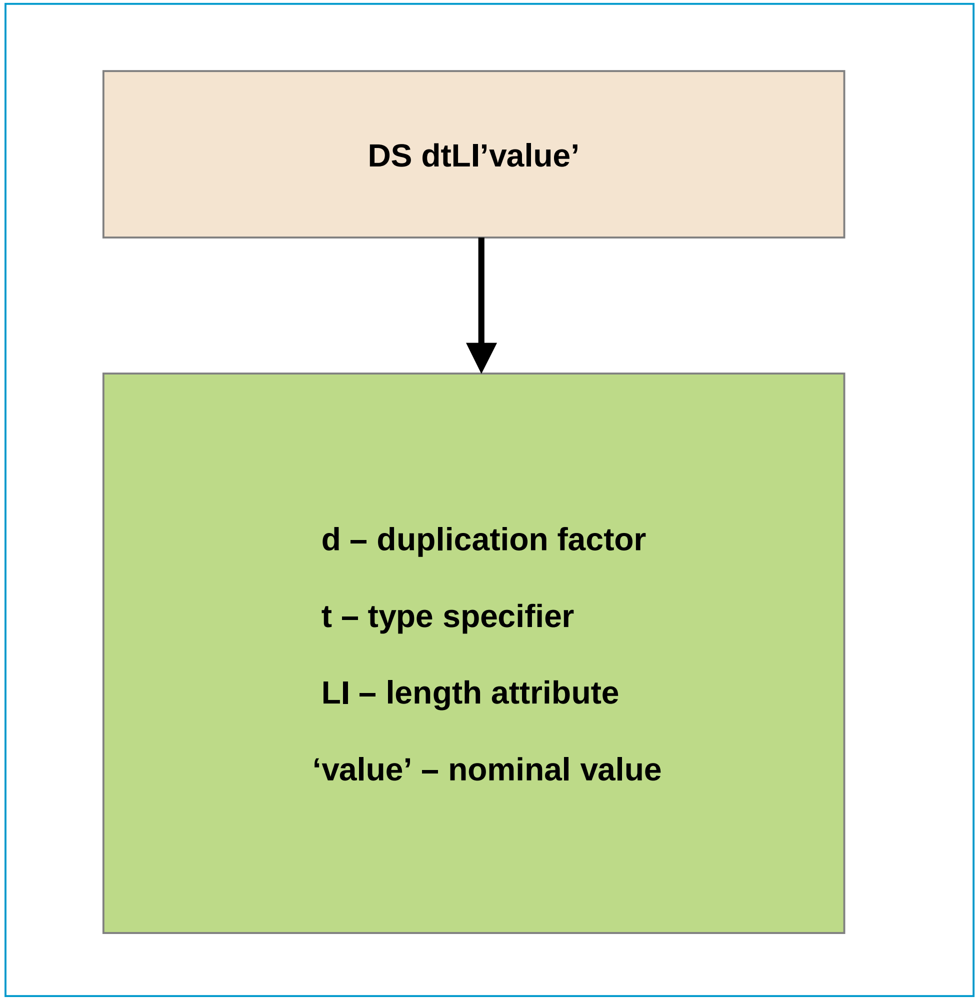


Components of a DS Statement

The type specifier must be supplied, while the other three components are optional.

The duplication factor and the length modifier has default values when not coded explicitly.

The nominal value field is not used, and is simply considered as documentation since it is defining storage, not a constant.



Note!

A common mistake is coding a nominal value in a DS statement, when a DC statement is intended. This is not flagged as an error by the Assembler, since the syntax is correct, but will likely produce an execution time error.

Continued…

Components of a DS Statement (cont'd)

Data Type	Type Specifier	Default Length	Default Alignment
Character	C	1	byte
Hexadecimal	X	1	byte
Binary	B	1	byte
Zoned Decimal	Z	1	byte
Packed Decimal	P	1	byte
Halfword	H	2	halfword
Fullword	F	4	fullword
Long Floating Point	D	8	doubleword

The type specifier indicates the type of data the field should contain. Each data type is specified by a single character identifier, and has a default length and alignment. The most common field types are indicated above.

Example:

To define a fullword called FWD1, you would specify the following statement as indicated: FWD1 DS F

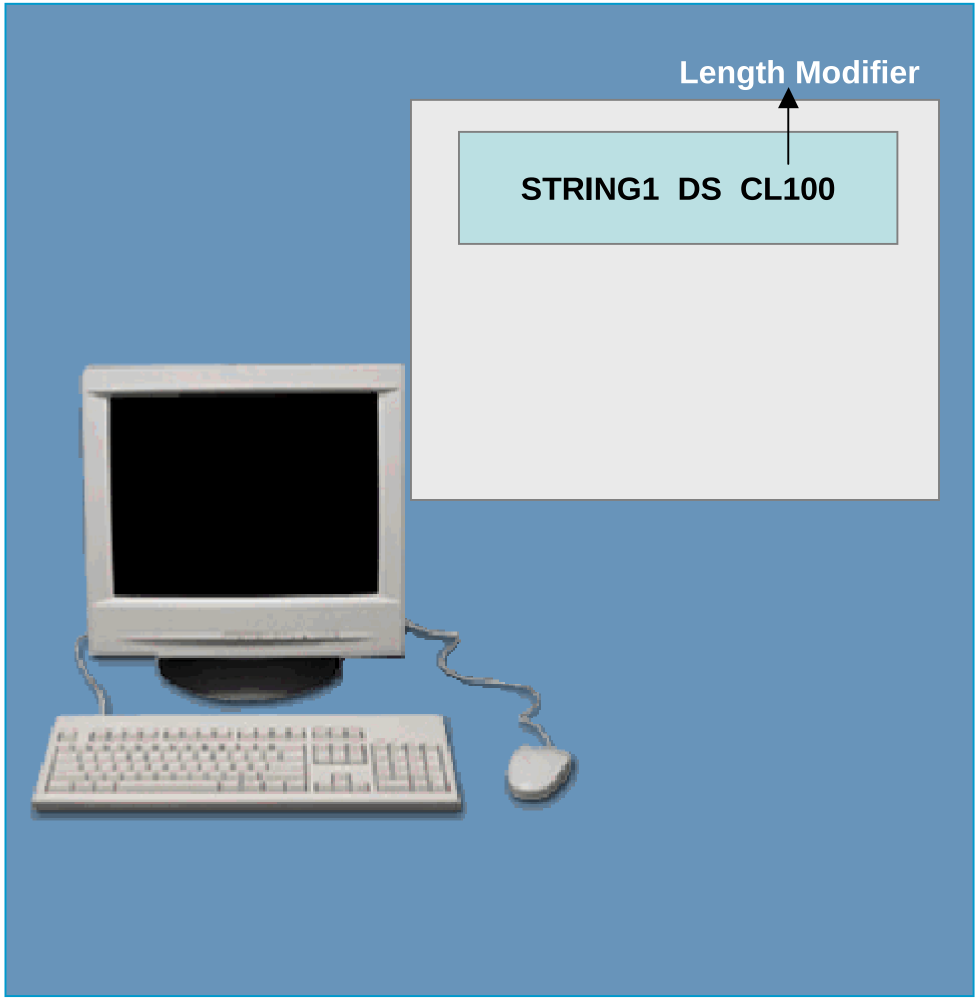
Length Modifier

What is the function of a length modifier?

Often you wish to define a field with a length different from a default length. To do this, you specify a length modifier. This consists of the letter L, followed by the desired field length in bytes. It is placed immediately following the type specifier.

To define a character field 100 bytes long, called STRING1, you would specify:

```
STRING1 DS CL100
```



Continued…

Length Modifier (cont'd)

For fields that are aligned on other than a byte boundary, specifying a length modifier removes the alignment guarantee, and reverts to byte alignment.

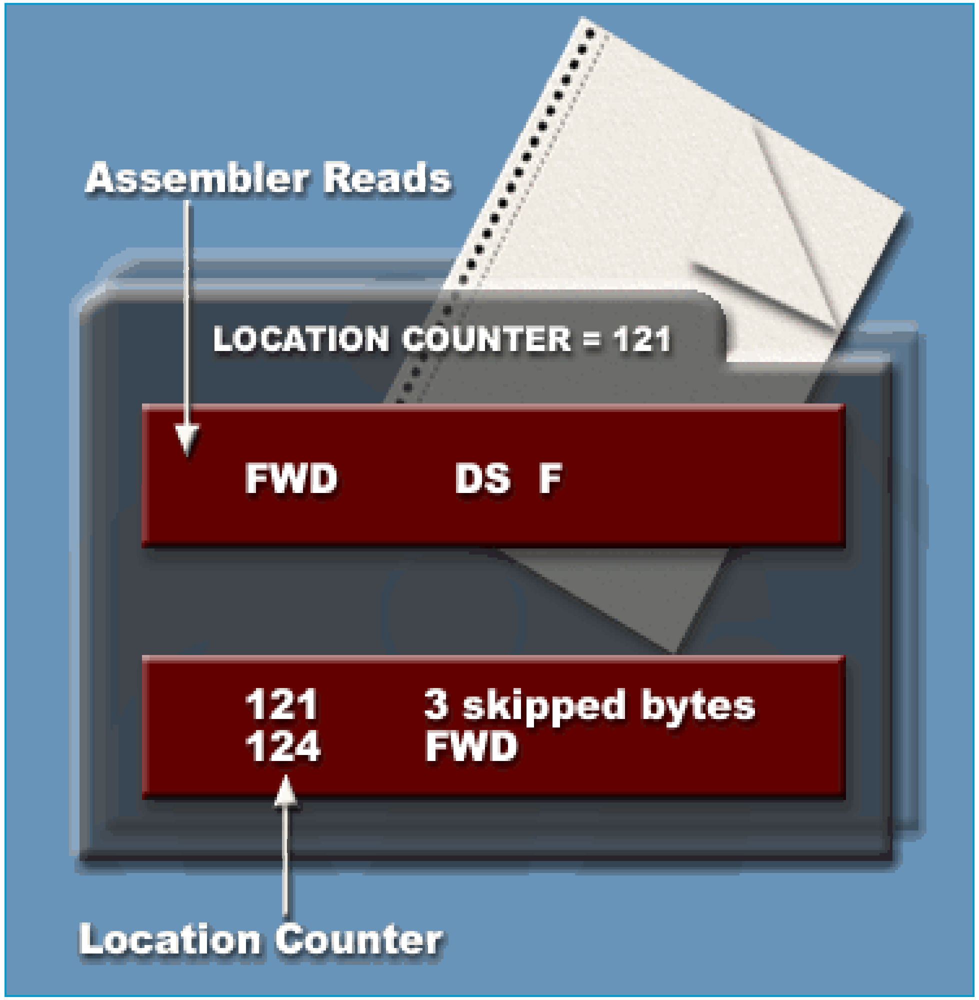
There is a guarantee that FWD2 would be aligned on a fullword boundary. There is no boundary alignment guarantee for FWD3. One way to force boundary alignment even if the definition does not guarantee it, is to precede the field with another statement that forces the desired alignment.

```
FWD2          DS  F
FWD3          DS  FL3
```

Location Counter

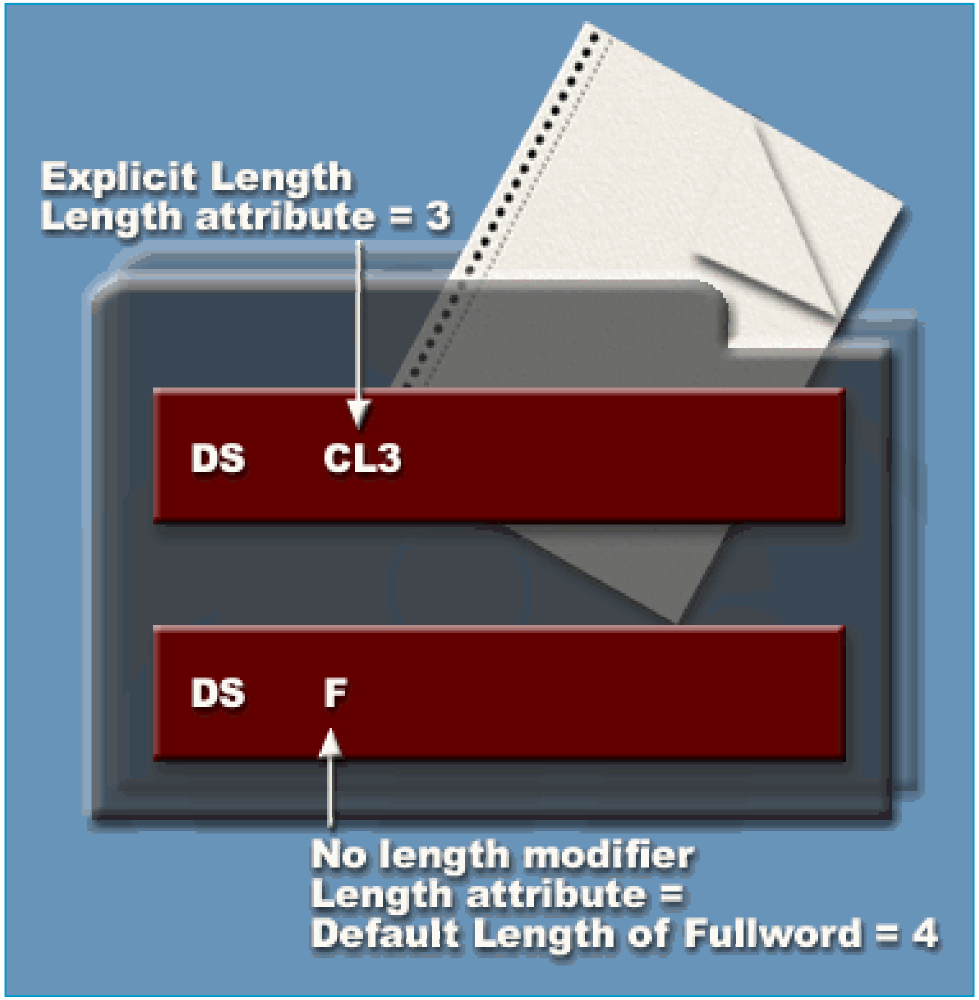
The Assembler assigns each new field you define immediately following the previous one, except when alignment is required.

The Assembler maintains a location counter (LC), which starts at zero at the beginning of the Assembly, and is increased by the length of each assembled instruction as the assembler proceeds. If the alignment is required, and the value of the location counter is not at an address that would provide the appropriate alignment, then the Assembler skips enough bytes to such an address that provides the required alignment. The skipped bytes are not initialized to any specific value.



Length Attribute

The value specified as the length modifier, or the default length if no length modifier is coded, is the length attribute of the field. The Assembler associates a length attribute to each field you define, and you may use it in assembling SS type Machine instructions, which reference the field.



Unit: Defining Data Topic: The Define Storage (DS) Instructions

The Duplication Factor

What is the duplication factor?

The duplication factor, if it is provided, is coded as the first part of the operand, preceding the type specifier. If the duplication factor is omitted, a default value of 1 is assumed. The duplication factor specifies the number of occurrences of the specified type and length specified.

1 – To define a program save area of 18 fullwords, we would code:
SAVEAREA DS 18F

2 – To define an array with 24 entries to hold hourly temperature readings (represented as halfwords) for one day we would code:
TEMPARR DS 24H

3 – To define an array to hold the marks of 100 students, each represented by a 2 byte packed decimal number we would code:
MARKS DS 100PL2

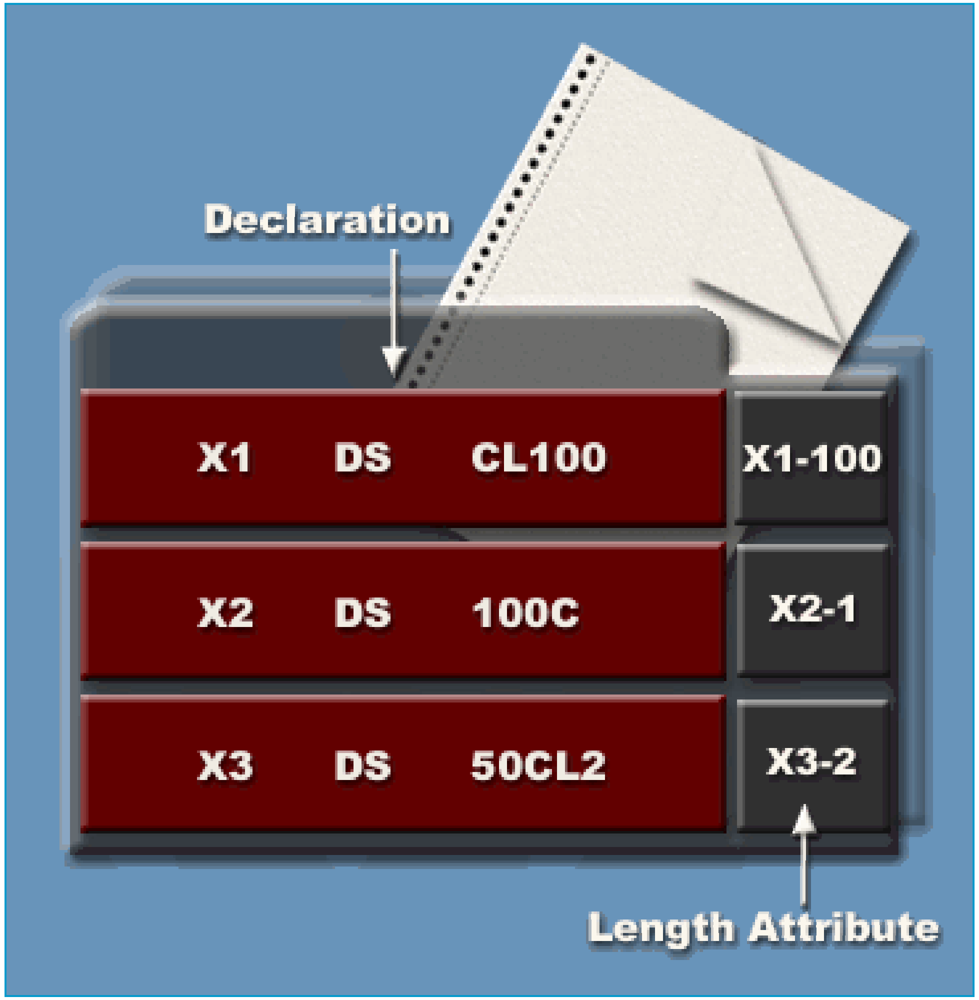
Note! The amount of storage allocated by a DS instruction is the product of the duplication factor and the length.

The Length Attribute

Why is the length attribute important?

Consider the following three declarations on the right. What is the difference between them?

They all allocate 100 bytes of storage, which is defined as character type. Are they identical in effect? Not quite. Remember that each data item as a length attribute assigned to it. The length attribute is the implicit or explicit length modifier. Thus the length attribute of the three declarations are as shown on the right:

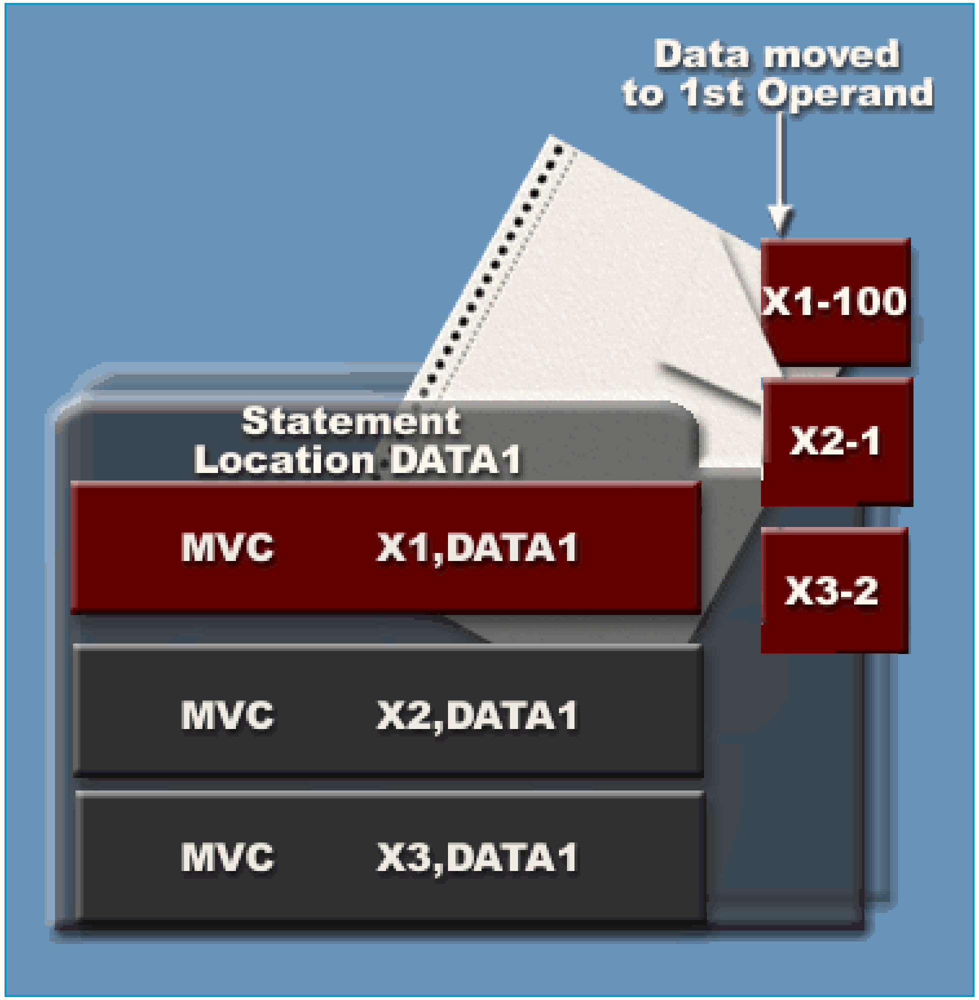


Continued…

The Length Attribute (cont'd)

What is the effect of the following 3 statements on the right?

All three of these statements move data from the location DATA1 to the destination (first operand) field. In the first case, 100 bytes of data are moved. In the second case, only 1 byte is moved and in the third case 2 bytes are moved.

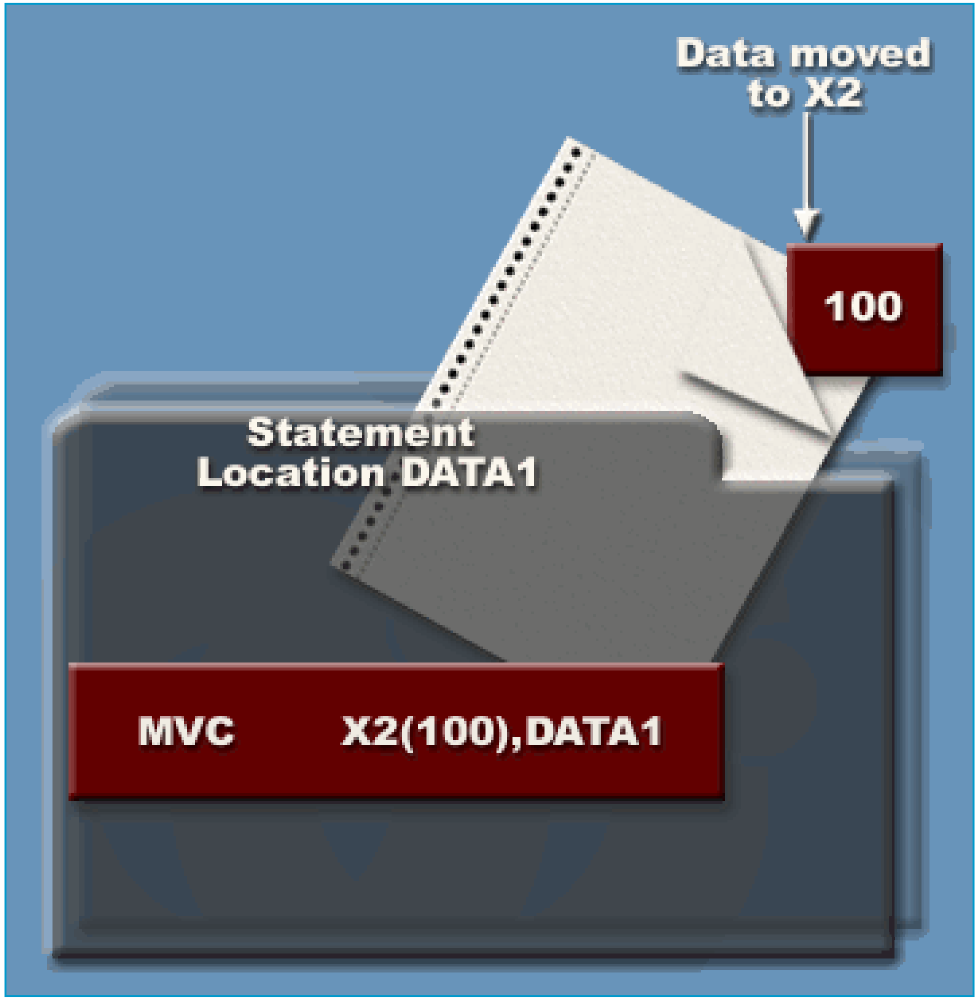


Continued…

The Length Attribute (cont'd)

To move 100 bytes of data from DATA1 to X2, the length would need to be coded explicitly in the MVC instruction. It would look like the statement on the right.

When an explicit length is omitted in an SS instruction, the length attribute of the destination field is used.

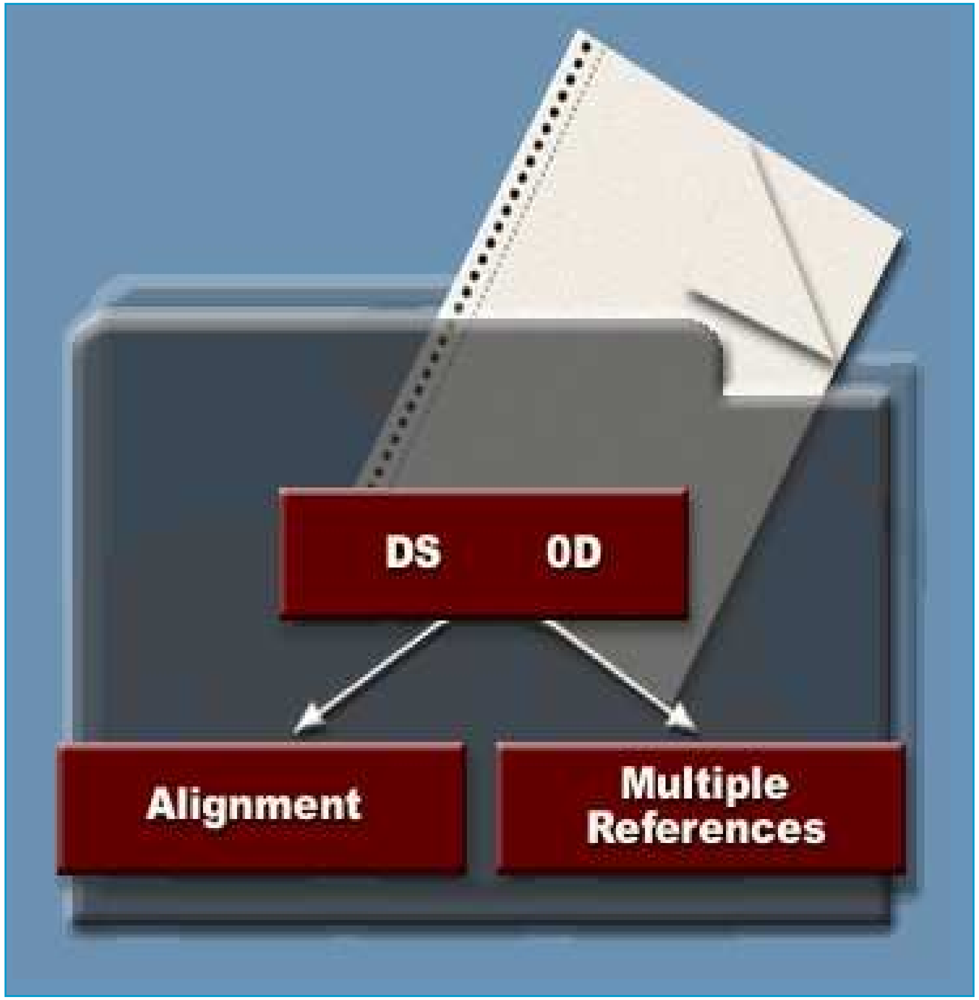


Use of Duplication Factors

Why are zero duplication factors used?

To specify a duplication of zero is a special situation of a data declaration. Why is this done?

Remember that the amount of storage allocated is the product of the duplication factor and the length. If the duplication factor is zero, so is the product of duplication factor and length, so no storage is allocated.


DS 0D

Alignment

Multiple References

Continued…

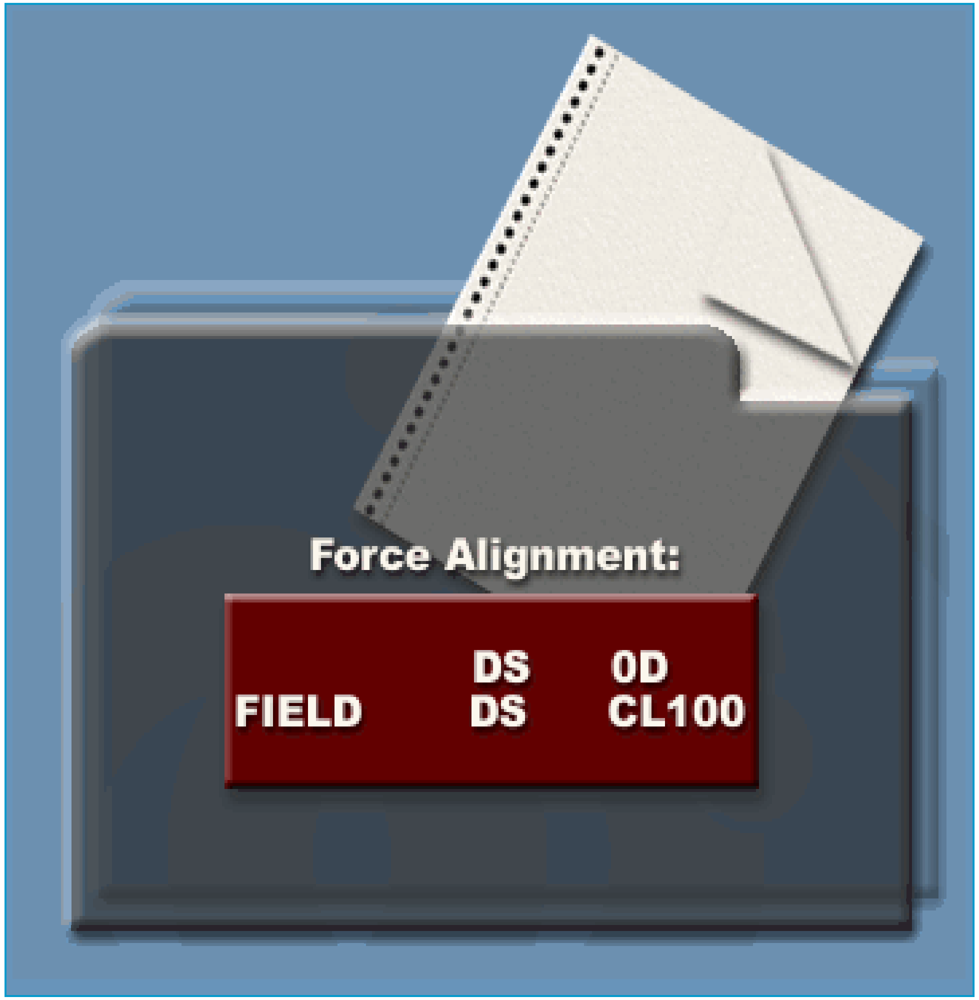
Use of Duplication Factors (cont'd)

Common reasons for using zero duplication factors:

One reason is to force alignment.

Suppose a 100 - byte character field is to be defined which is aligned on a doubleword boundary. The easiest way to do this is to define a doubleword (forcing doubleword alignment) immediately followed by the character field. Since the doubleword is not actually used for anything, other than to force alignment, it is defined with a duplication factor of zero. It does not even have to be given a name, since it will not be used. Its only purpose is to cause the Assembler to skip enough bytes to move to the doubleword boundary.

Thus the definition looks as it does on the right.



Force Alignment:

```
         DS    0D
FIELD    DS    CL100
```

Continued…

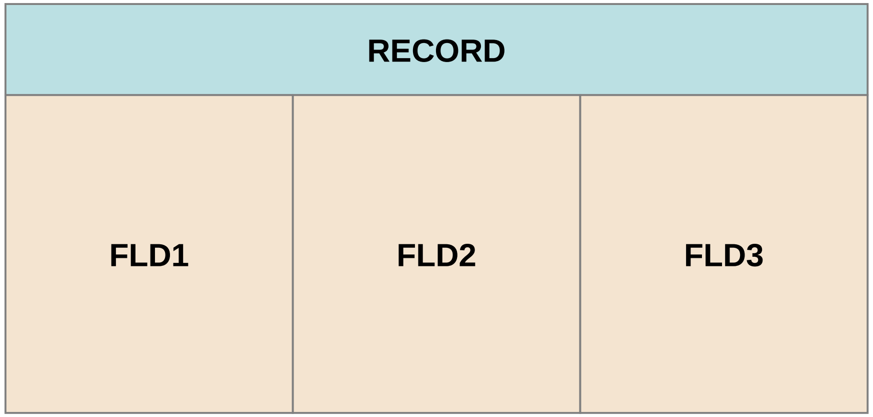
Use of Duplication Factors (cont'd)

In the other situation a zero duplication factor is often used is when the same piece of storage is defined in two or more different ways.

Suppose an area of storage is to be set up to hold a record that will be read into our program from disk. The record is 120 bytes in length, but suppose this record contains 3 fields, each 40 bytes in length. When reading the record the programmer wishes to treat it as a single 120-byte field. After it is read, the programmer wants to process its fields separately. It can be accomplished using a zero duplication factor on the first definition.

The definition is shown on the right.

MULTIPLE REFERENCES TO DATA		
RECORD	DS	0CL120
FLD1	DS	CL40
FLD2	DS	CL40
FLD3	DS	CL40



Continued…

Use of Duplication Factors (cont'd)

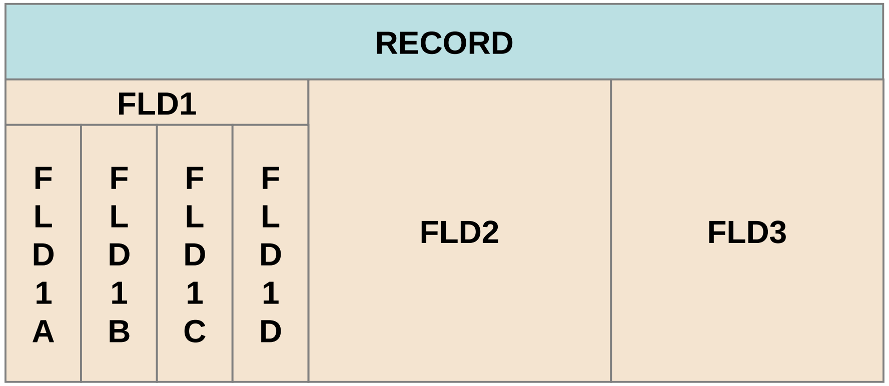
The RECORD is defined with a zero duplication factor, the actual space assigned to it is 0 * 120 = 0.

The length attribute of RECORD is 120. FLD1 has the same location as RECORD, but has a length attribute of 40. The space assigned by the Assembler is 1 * 40 = 40.

The address of FLD2 is 40 higher than the address of FLD1. FLD2 also has a length attribute of 40 as does FLD3.

The fields can be broken down even further. FLD1 could consist of 4 smaller fields.

MULTIPLE REFERENCES TO DATA		
RECORD	DS	0CL120
FLD1	DS	0CL40
FLD1A	DS	CL5
FLD1B	DS	CL15
FLD1C	DS	CL8
FLD1D	DS	CL12
FLD2	DS	CL40
FLD3	DS	CL40



RECORD

FLD1

FLD1A

FLD1B

FLD1C

FLD1D

FLD2

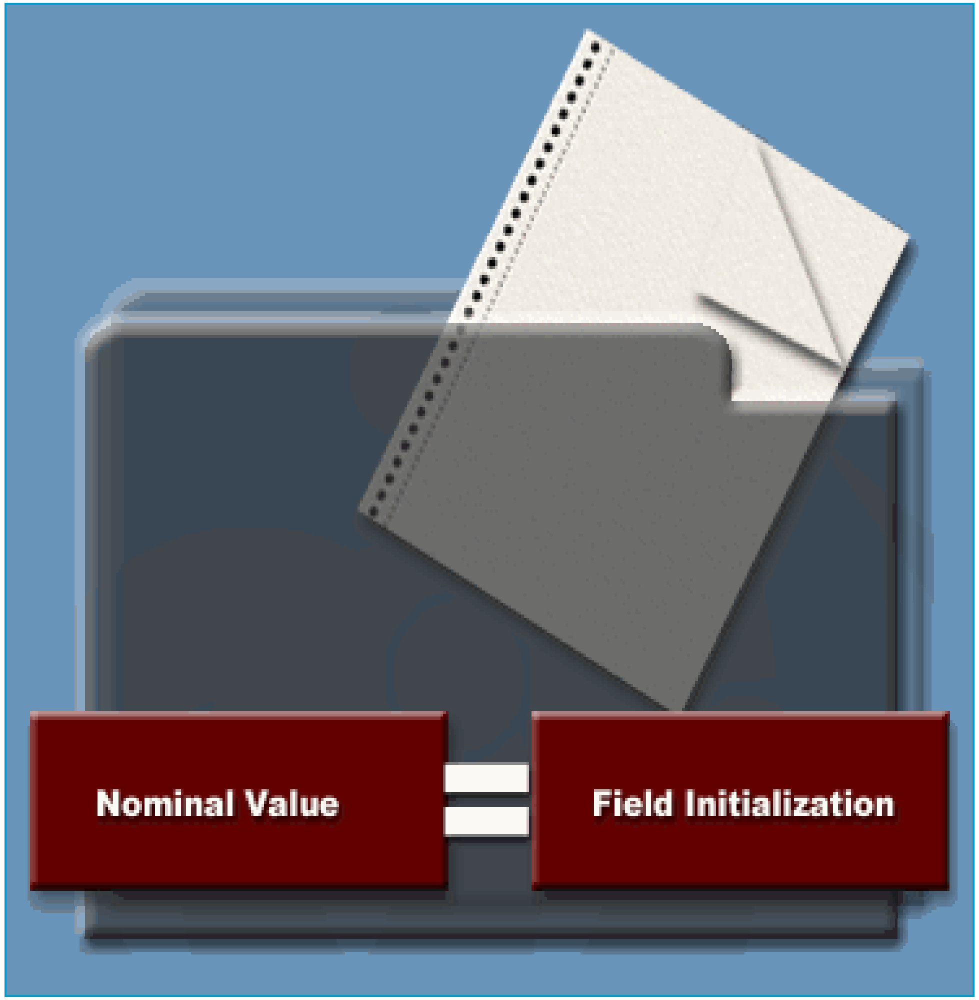
FLD3

Define Constant (DC) Instruction

What is Define Constant (DC) Instruction?

The Define Constant (DC) instruction is used to define constants. In fact the term constant is somewhat misleading since there is no guarantee that the storage set up by a DC Assembler instruction will remain constant throughout program execution. In fact, what DC does is to define storage and initialize it to the value you specify in the DC instruction.

The operand field of DC contains the same 4 components as the DS instruction, but the nominal value field is compulsory. The nominal value specifies the value to which the field is initialized.

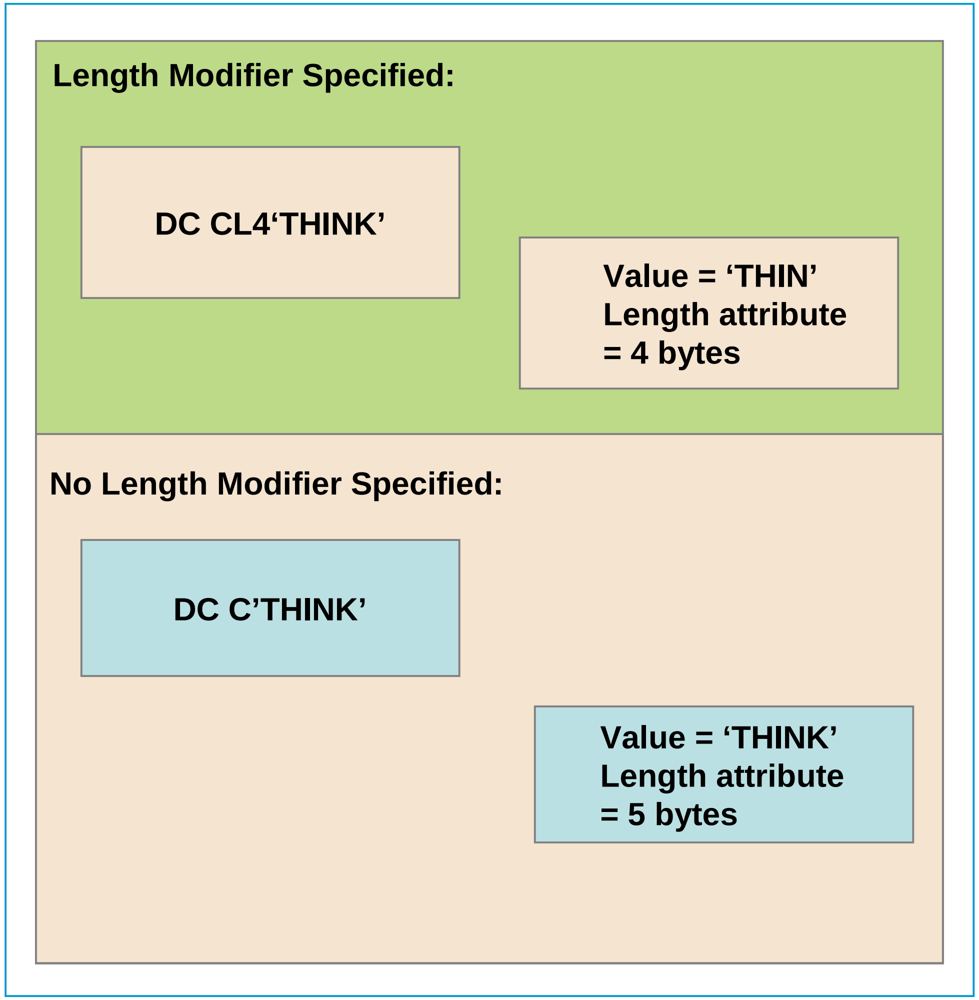


Length of a Field

The length of a field defined by a DC statement can be established in one of two ways. If a length modifier is specified in the definition, that length is the one used.

If no length modifier is specified, then the length required to hold the nominal value is used. If an explicit length is specified, and it does not agree with the length of the nominal value, then the nominal value is either padded or truncated to fit the explicitly specified field length.

The way that padding or truncation is done depends on the field type. For character constants, the nominal value is left justified and padded on the right with blanks if it is shorter than the field length. It is truncated on the right if it is too long. The nominal value is specified as a series of characters.



Length Modifier Specified:

DC CL4‘THINK’

Value = ‘THIN’ Length attribute = 4 bytes

No Length Modifier Specified:

DC C’THINK’

Value = ‘THINK’ Length attribute = 5 bytes

Character Value

The Assembler converts the character value specified in the definition into EBCDIC code. In the following examples, we will represent the field contents (EBCDIC) in hex notation.

With no length modifier the length is determined by the nominal value

Definition			Field Contents
CFLD1	DC	C'ABC'	C1C2C3

Explicit length of 5, the nominal value is left justified and right padded with blanks

Definition			Field Contents
CFLD2	DC	CL5'ABC'	C1C2C34040

Explicit length of 2, the nominal value is left justified and right truncated

Definition			Field Contents
CFLD3	DC	CL2'ABC'	C1C2

Duplication factor of 2, with length determined by the nominal value

Definition			Field Contents
CFLD4	DC	2C'ABC'	C1C2C3C1C2C3

Continued…

Character Value (cont'd)

The maximum length of a character constant is 256 bytes.

Duplication factor of 2 and explicit length of 3

Definition			Field Contents
CFLD5	DC	2CL3'ABC'	C1C2C3C1C2C3

Explicit length of 10, the nominal value is left justified and right padded with blanks

Definition			Field Contents
CFLD6	DC	CL10'*'	5C404040404040404040

Duplication factor of 10, length of 1 is determined by the nominal value

Definition			Field Contents
CFLD7	DC	10C'*'	5C5C5C5C5C5C5C5C5C5C

Hexadecimal Constants

When do you use hexadecimal constants?

Hexadecimal constants, like other numeric constants, are right justified in their field, padded on the left with zeroes or truncated on the left if needed. The nominal value is specified as a series of hexadecimal characters. The examples below illustrate the use of hexadecimal constant definitions.

With no length modifier the length is determined by the nominal value

Definition			Field Contents
XFLD1	DC	X'3C2F'	3C2F

Explicit length of 3 is longer the nominal value. Field is right justified and left padded with zeros.

Definition			Field Contents
XFLD2	DC	XL3'3C2F'	003C2F

Explicit length of 1 is shorter then the nominal value. Field is left truncated.

Definition			Field Contents
XFLD3	DC	XL1'3C2F'	2F

Field must contain an even number of nibbles. Leading zero is inserted.

Definition			Field Contents
XFLD4	DC	X'ABC'	0ABC

Continued…

Hexadecimal Constants (cont'd)

With data types other than character, multiple values can be defined in a single DC statement, by separating the values with commas.

Duplication factor of 2 and and padded to an even number of nibbles.

Definition	**Field Contents**
XFLD5 DC 2X'ABC'	**0ABC0ABC**

Duplication factor of 3 and left truncated to 3 bytes.

Definition	**Field Contents**
XFLD6 DC 3XL3'1234567AB'	**4567AB4567AB4567AB**

Two values, with each length determined from the length of the nominal value.

Definition	**Field Contents**
XFLD7 DC X'AB,CDE'	**AB0CDE**

Binary Constants

When do you use binary constants?

As a numeric type, binary constants are right justified, truncated on the left or padded on the left with zeroes. The nominal value is specified as a string of bits (ones and zeroes).

Both binary and hexadecimal constants have a maximum length of 256.

With no length modifier, length is determined by the nominal value. Three leadings zero bits are added to form a complete byte.

Definition			Field Contents (Hex)
BFLD1	DC	B'10111'	17

Length modifier of 2 determines the field length.The field is padded on the left with zeros.

Definition			Field Contents
BFLD2	DC	BL2'10111'	0017

Length modifier of 1 determines the field length. The nominal value is right justified and left truncated.

Definition			Field Contents (Hex)
BFLD3	DC	BL1'10101010101'	55

Duplication factor of 2, left padded with zeros to form a complete byte.

Definition			Field Contents (Hex)
BFLD4	DC	2B'101101'	2D2D

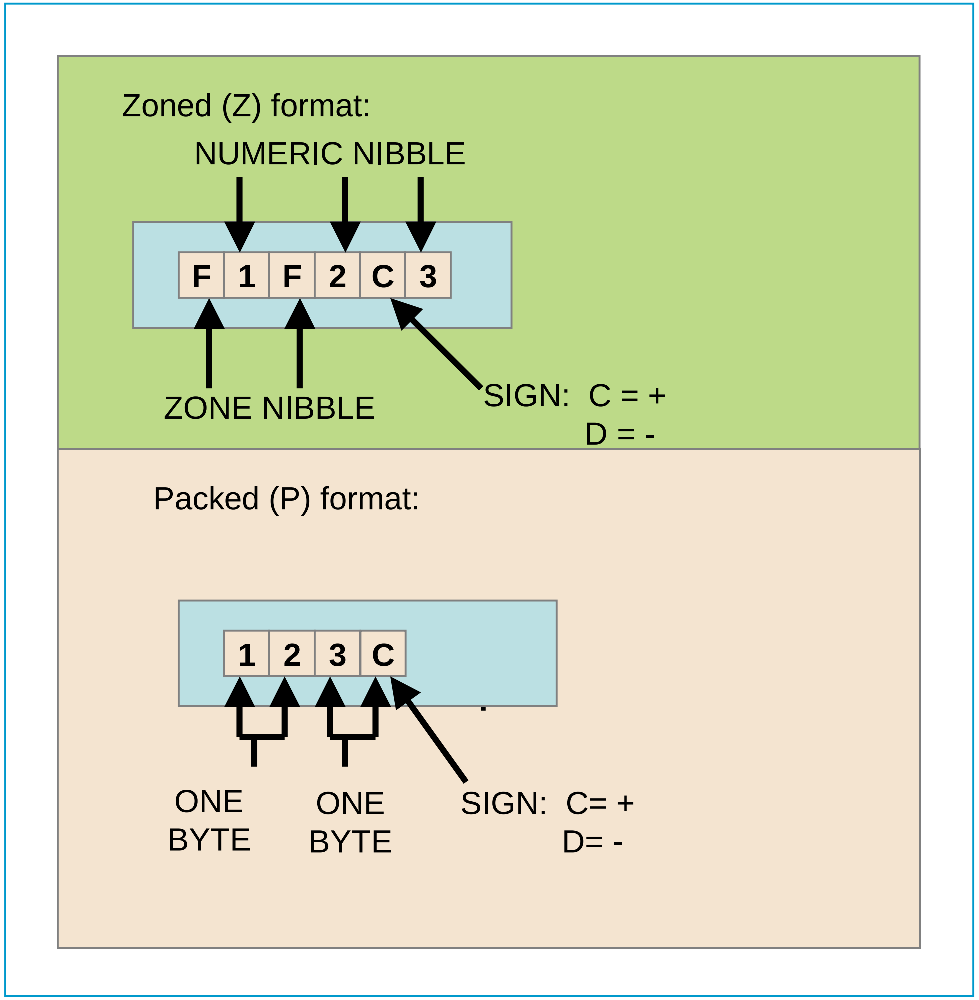
Zoned Decimal Format

What are the two formats of decimal numbers?

There are two formats of decimal numbers, zoned decimal and packed decimal.

Zoned decimal is represented one digit per byte, with each byte consisting of a zone nibble and a numeric nibble.

In packed decimal format, each byte contains two numeric nibbles, except the rightmost, which contains one numeric nibble and the sign nibble.



Note! The zone nibble is F in all nibbles except the rightmost, where it is a sign (usually C for + and D for -).

Continued…

Zoned Decimal Format (cont'd)

As with other numeric types, decimal constants are right justified, padded on the left with zeroes or truncated on the left.

Some examples of zoned decimal constants are shown on the right.

With no length modifier, length is determined by the nominal value.

Definition			Field Contents
ZFLD1	DC	Z'12345'	F1F2F3F4C5

Explicit length of 6, so field is left padded with a zero.

Definition			Field Contents
ZFLD2	DC	ZL6'12345'	F0F1F2F3F4C5

Explicit length of 4, so field is left truncated.

Definition			Field Contents (Hex)
ZFLD3	DC	ZL4'12345'	F2F3F4C5

Three separate constants.

Definition			Field Contents (Hex)
ZFLD4	DC	Z'1,-2,3'	C1D2C3

Duplication factor of 2.

Definition			Field Contents
ZFLD5	DC	2ZL3'125'	F1F2C5F1F2C5

Unit: Defining Data Topic: The Define Constant (DC) Instructions

Packed Decimal

Although decimal numbers are integers, you can specify nominal values that have an embedded decimal point. In such cases, the decimal point is ignored. The technique can be useful for documentation purposes, if the numbers are being used to represented a quantity with a decimal, such as dollars and cents. The hardware treats the value as an integer, but the programmer can provide code to present the number with a decimal point when it is printed.

The maximum length of a decimal constant is 16 bytes.

With no length modifier, length is determined by the nominal value.

Definition			Field Contents
PFLD1	DC	P'12345'	12345C

Explicit length of 4, so left padded with a zeros.

Definition			Field Contents
PFLD2	DC	PL4'12345'	0012345C

Explicit length of 2, so left truncated.

Definition			Field Contents (Hex)
PFLD3	DC	PL2'12345'	345C

Three constants, last two padded.

Definition			Field Contents (Hex)
PFLD4	DC	P'1,37,-4892'	1C037C04892D

Decimal point is ignored, left padded with zeros.

Definition			Field Contents
PFLD5	DC	PL4'39.45'	0003945C

Fixed Point Numbers

Fixed-point numbers (halfwords and fullwords) are represented internally in true binary format for positive numbers and in two’s complement form for negative numbers. The nominal value is coded in decimal.

Constant is 2 bytes (1 halfword) long, represented in true binary

Definition			Field Contents
HFLD1	DC	H'10'	000A

Constant is 2 bytes (1 halfword) long, represented in two's complement form.

Definition			Field Contents
HFLD2	DC	H'-5'	FFFB

Duplication factor of 2.

Definition			Field Contents (Hex)
HFLD3	DC	2H5'5'	00050005

Three constants.

Definition			Field Contents
HFLD4	DC	H'50,-10,160'	0032FFF600A0

Continued…

Fixed Point Numbers (cont’d)

The nominal value is coded in decimal.

Constant is 4 bytes (1 fullword) long, represented in two's complement.

Definition			Field Contents
FFLD1	DC	F'255'	000000FF

Constant is 4 bytes (1 fullword) long, represented in two's complement.

Definition			Field Contents
FFLD2	DC	F'-100'	FFFFFF9C

Duplication factor of 3.

Definition			Field Contents
FFLD1	DC	3F'0'	000000000000 0000000000000

Four constants.

Definition			Field Contents
FFLD4	DC	FL'31,28,31,30'	0000001F000000 1C0000001F0000001E

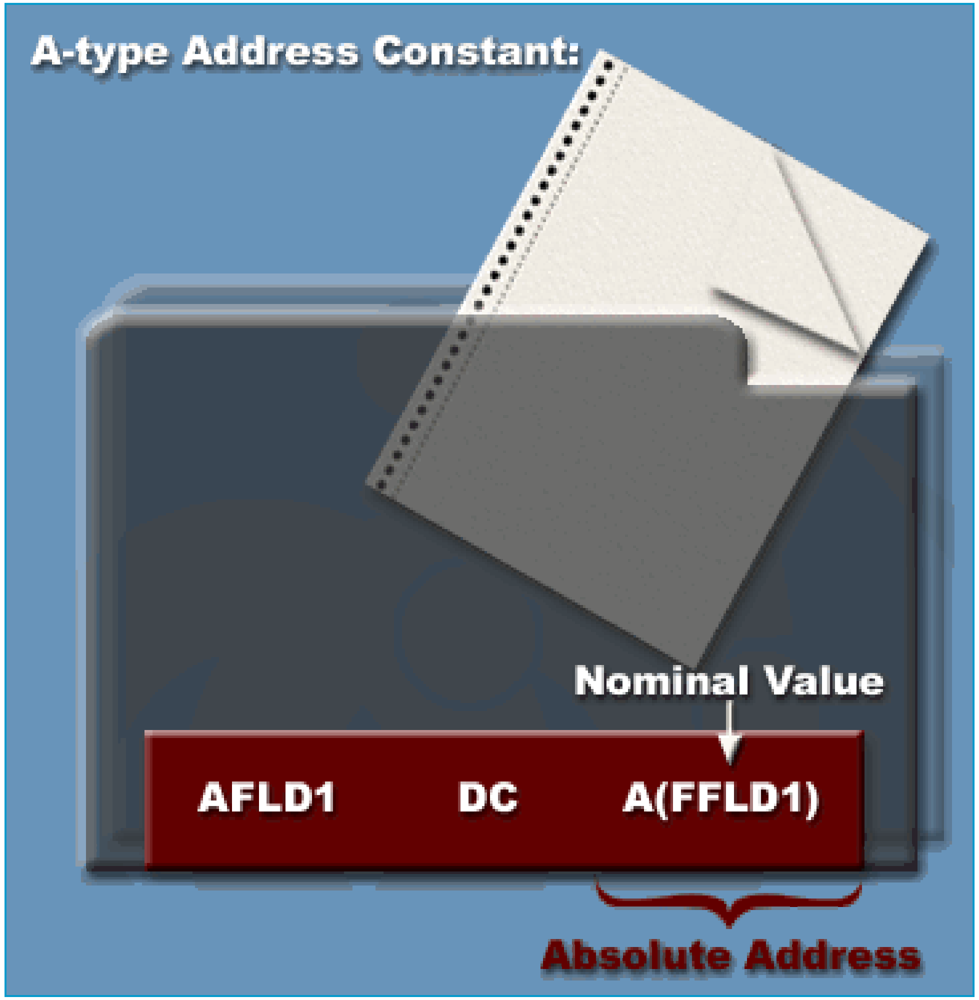
Address Constant

What is an Address Constant?

The final type of constant is called an address constant. There are several types of address constants, but only one will be discussed. An A-type address constant is 4 bytes long by default and is aligned on a fullword boundary. An A-type address constant contains the absolute address of its operand. The nominal value of an address constant is specified within parentheses, rather than quotes.

To have the Assembler generate an address constant containing the address of FFLD1, you would specify:

```
AFLD1  DC  A(FFLD1)
```



Literals

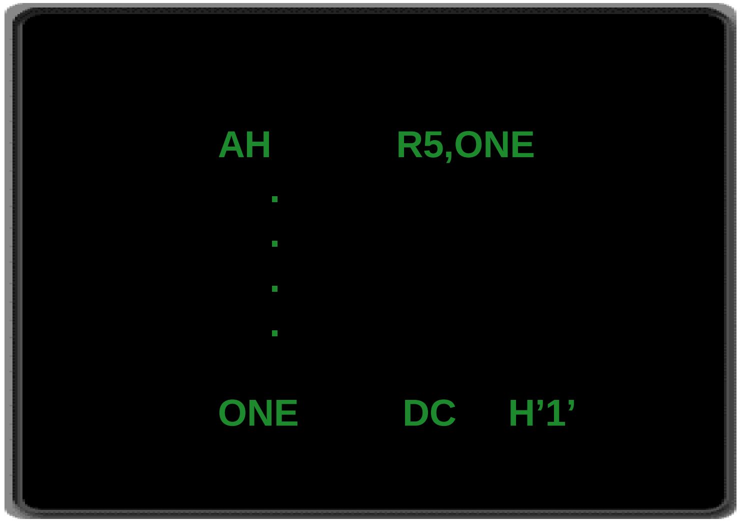
What are Literals?

Literals are used as a shortcut in Assembler Language programming.

Suppose you wished to add 1 to the fixed-point value in GPR 5.

One way to do this would be to be to code a constant with a value of 1, and then add that constant to the register. An example of the code is shown on the right.

Without Literals:



```
AH        R5,ONE
     .
     .
     .
     .
ONE       DC    H’1’
```

Continued…

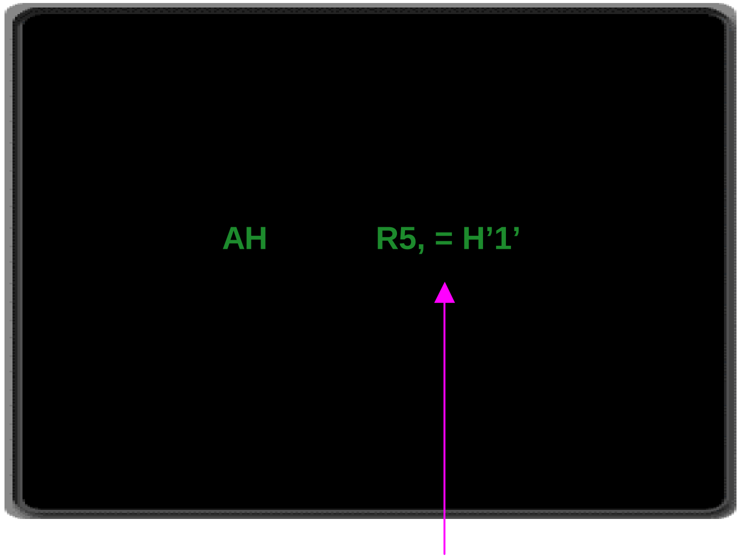
Literals (cont'd)

What are the uses of literals?

Using a literals allows you to eliminate the separate definition of the constant. A literal is represented as an equal sign, followed by a value that would be valid in the operand field of a *DC instruction*.

In the example, =H'1' is the literal value. What actually happens when you code a literal is that the assembler sets up a constant, at some place in storage (called the literal pool) and places the address of that constant in the operand field of the *AH Instruction*.

Without Literals:



```
AH          R5, = H'1'
```

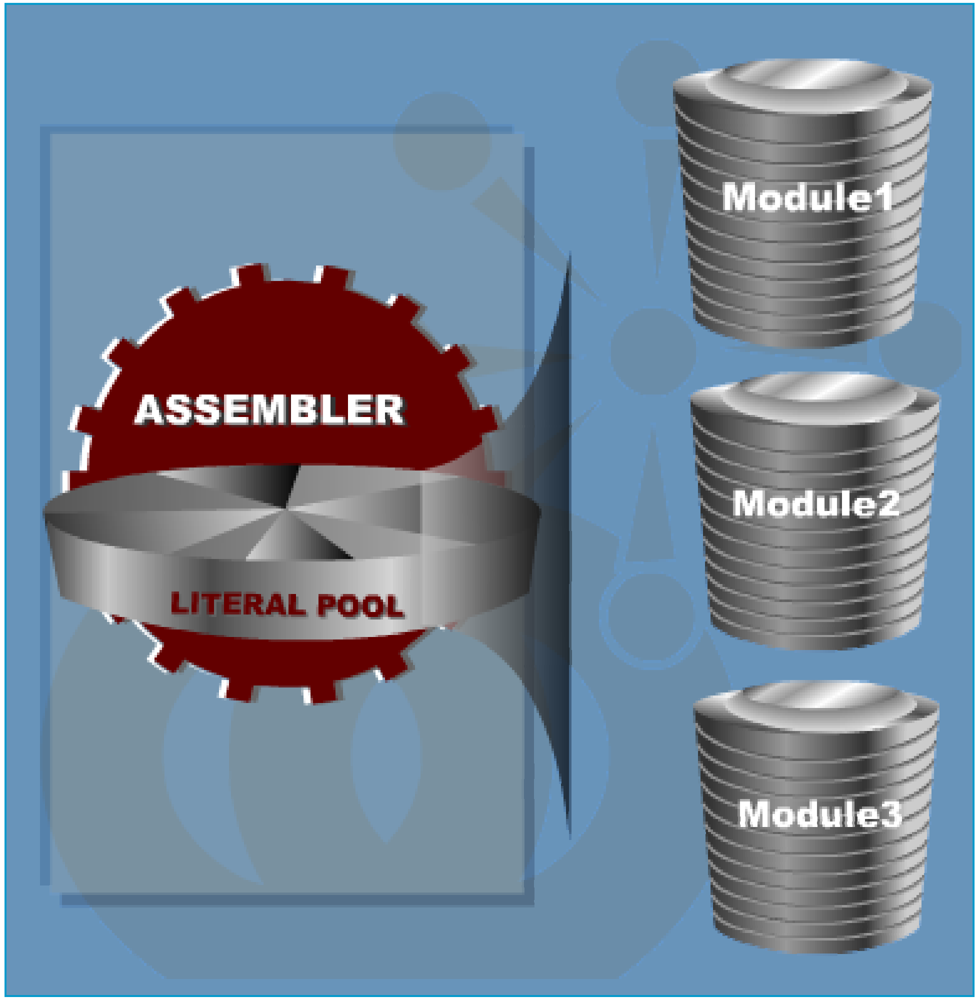
Literal Value

Unit: Defining Data Topic: Literals

Addressability

The Assembler creates an area called literal pool and places all literals into that area. The assembler provides the programmer with some control over where that literal pool is created. Why should we care? One reason is that some programs are composed of *multiple modules,* each of which establishes it own *addressability*.

A value defined in one module may not be addressable by code in another module. Each module should be self-contained. Unless we tell the Assembler differently, it will create a single literal pool for all the modules in the assembly. This is likely to produce addressability problems.

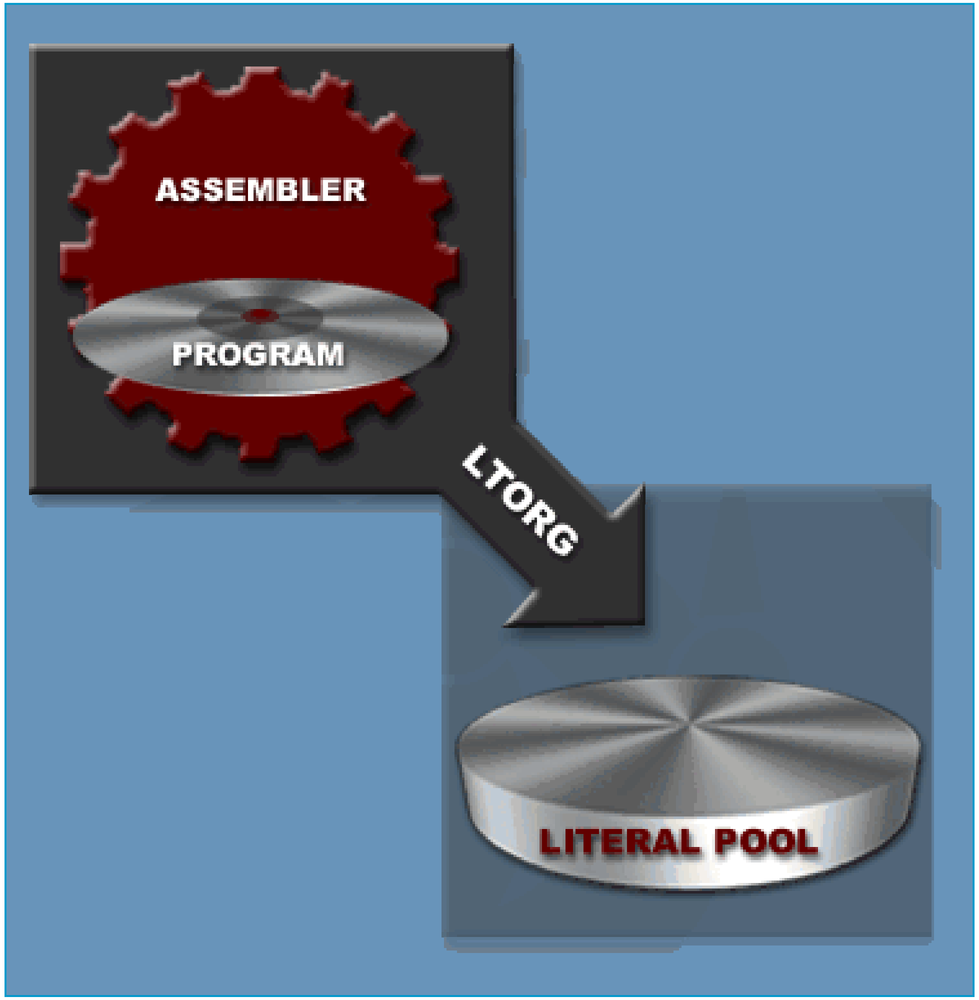


Functions of LTORG Instruction

What is the function of the LTORG?

The assembler instruction that directs the location of the literal pool is *LTORG*. It does not have any operands. If you do not use any *LTORG* instructions in your program, the literal pool is placed at the end of the *first control section (CSECT)* of your program.

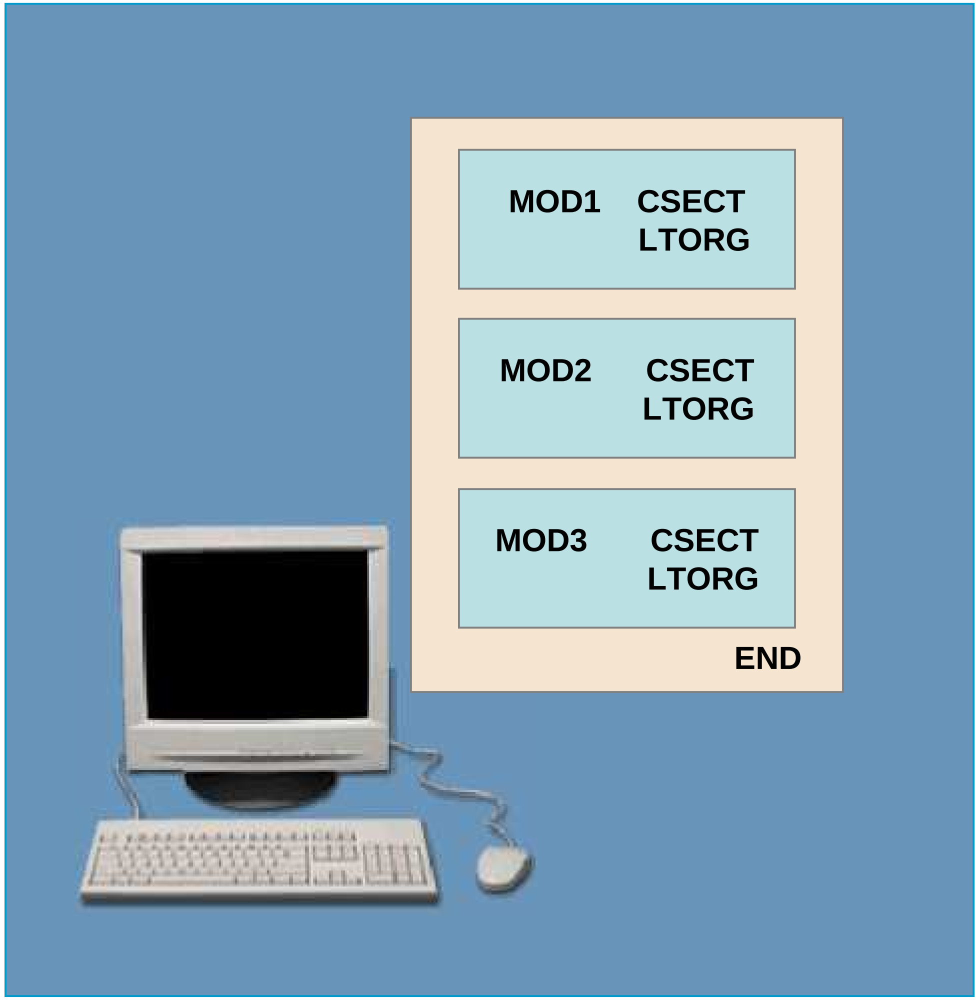
When the *assembler* encounters a *LTORG* instruction within a program, it creates a literal pool at that point, containing all the literals specified since the last *LTORG* instruction, or since the beginning of the program, if this is the first *LTORG*.



Continued…

Functions of LTORG Instruction (cont'd)

To avoid addressability problems in multi-modular programs, you should code a LTORG instruction as the last instruction of each control section in the program. So, each CSECT will have its own literal pool at the end of the module.



Processing Input and Output Data

The main purpose of most assembler programs is to read in data (input), process it, and then print out the resulting data (output).

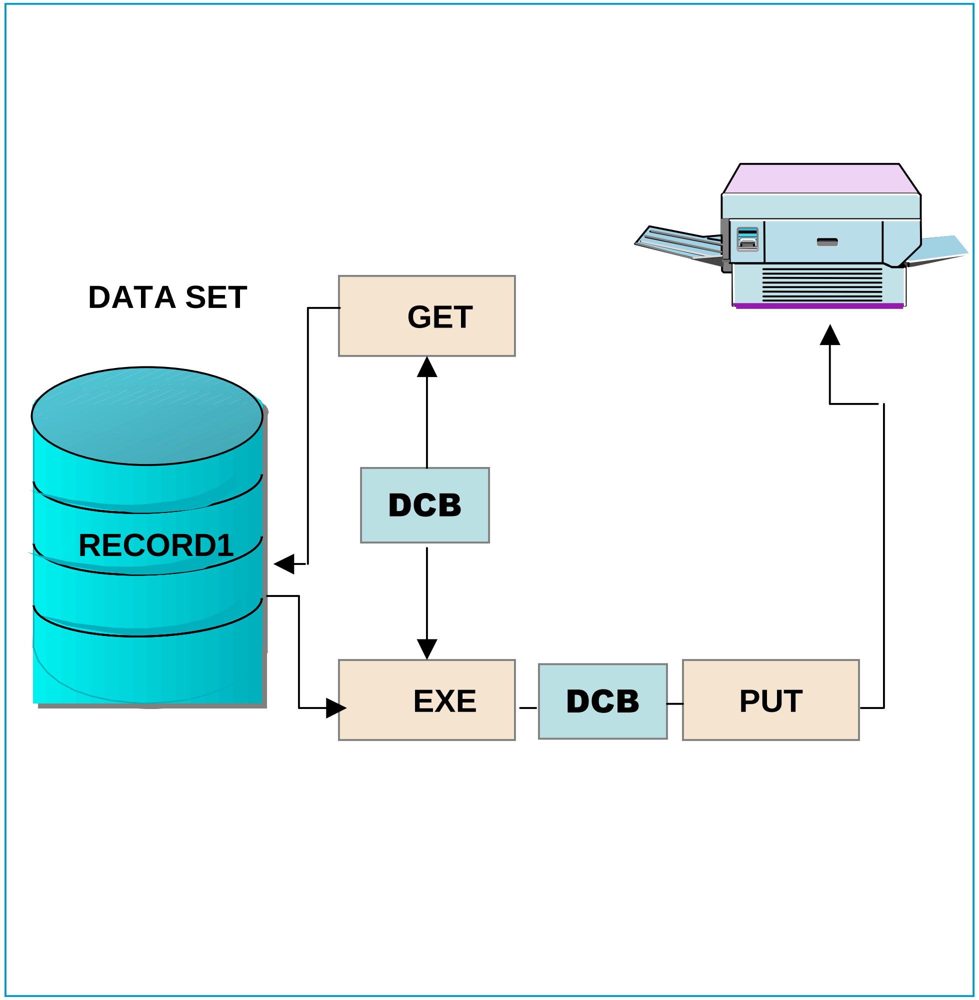
The major steps for processing input and output data in a program are shown. I/O macros are used to accomplish all of these tasks, except for the definition of I/O areas, which is done with assembler instructions.

I/O PROCESSING TASKS	MVS MACRO
1. Define the data set	DCB
2. Define I/O areas	DS,DC
3. Make the data set available	OPEN
4. Read input data	GET
5. Write output data	PUT
6. Free the data set	CLOSE

Elements of Defining Data Sets

Every Assembler program that processes I/O must provide information to the Assembler about the data sets to be processed. The data is organized sequentially and processed by the DCB (Data Control Block) macro in the same manner.

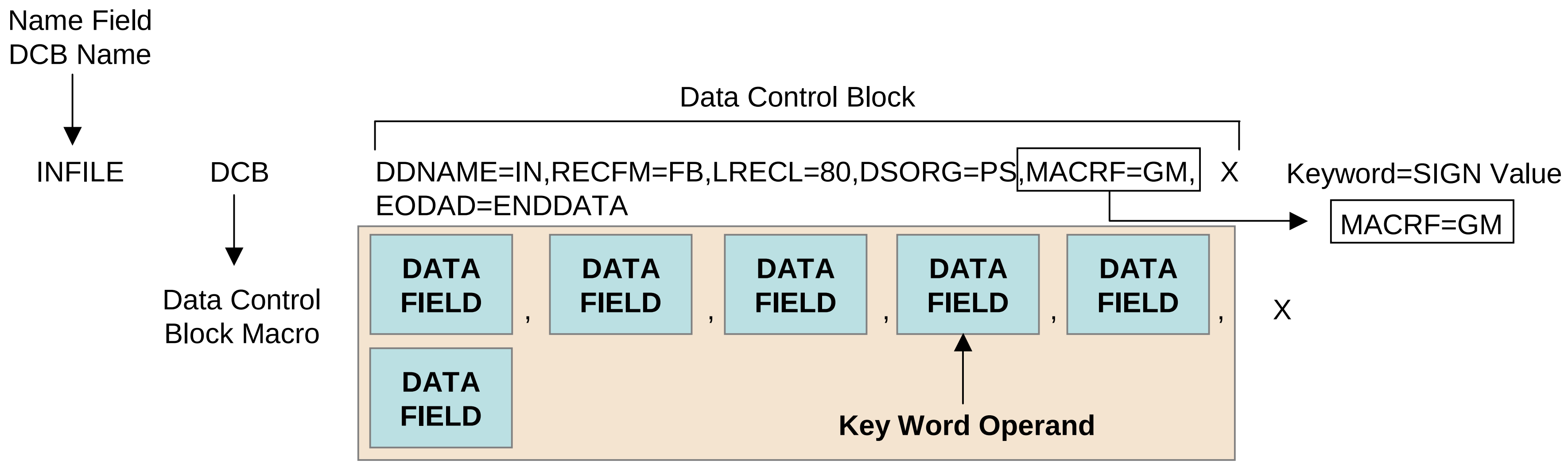
The DCB macro instruction generates a large block of constants in order to process the data set. The block of constants is a set of data fields (or parameters) that control the look of the data set being processed.



Continued…

Unit: Basic MVS I/O Facilities Topic: Defining Data Sets

Elements of Defining Data Sets (cont’d)



A DCB is a collection of data fields that contain information concerning the data set, including:

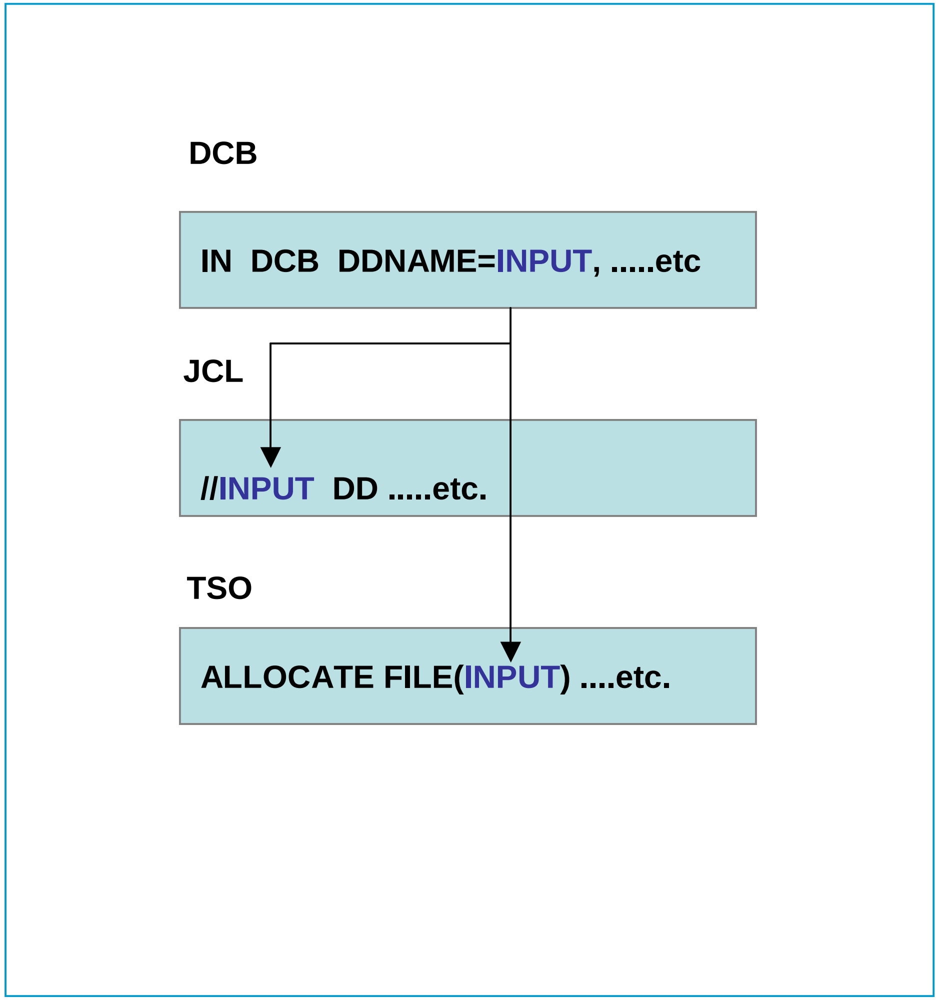
- The size and format of the records
- The I/O macros used to process the data
- The name and the current status of the data set

Parameters

What are the parameters used in DCBs?

A variety of keyword parameters are used with input and output DCBs.

The DDNAME parameter is used as a link to the external file this DCB represents. The value of the DDNAME parameter must match the DDNAME on the JCL DD statement for batch execution, or the FILE parameter on the ALLOCATE TSO statement for interactive execution.



DCB

```
IN  DCB  DDNAME=INPUT, .....etc
```

JCL

```
//INPUT  DD .....etc.
```

TSO

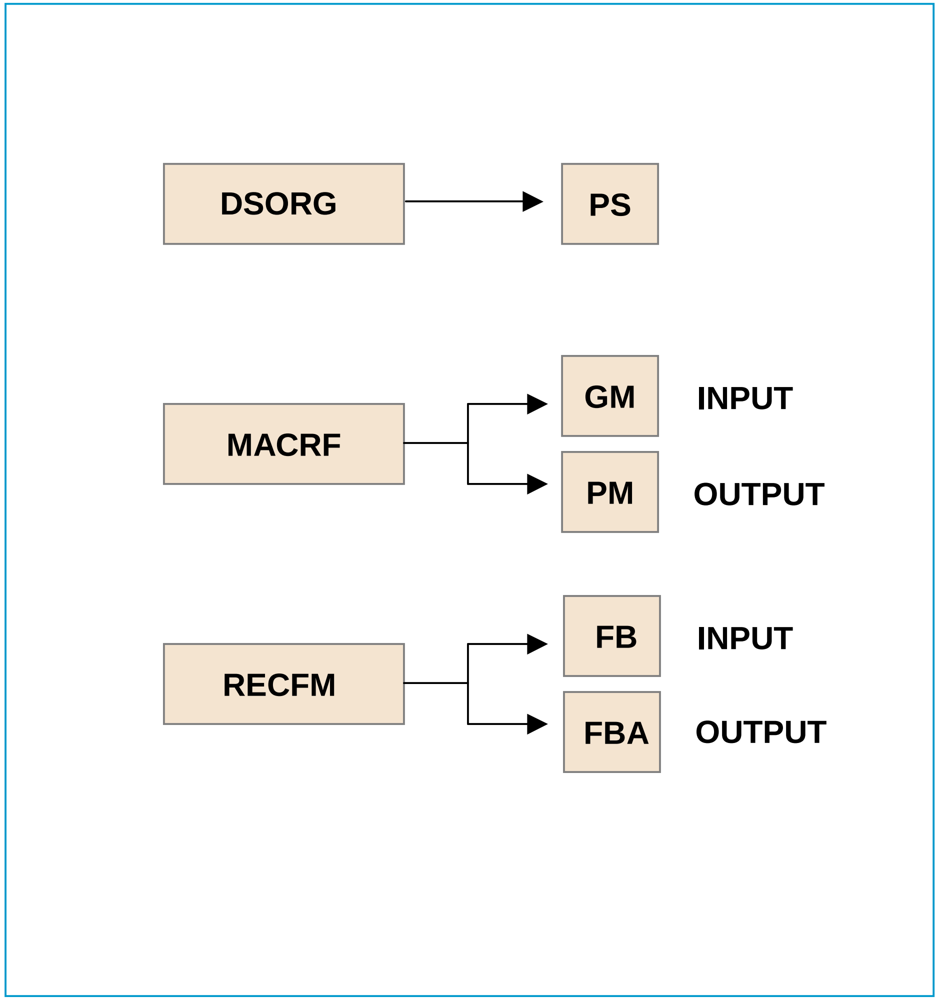
```
ALLOCATE FILE(INPUT) ....etc.
```

Kinds of Parameters

The **DSORG** parameter specifies the organization of the data set represented by this DCB. In the examples in this course, the parameter is always PS, which is an acronym for Physical Sequential.

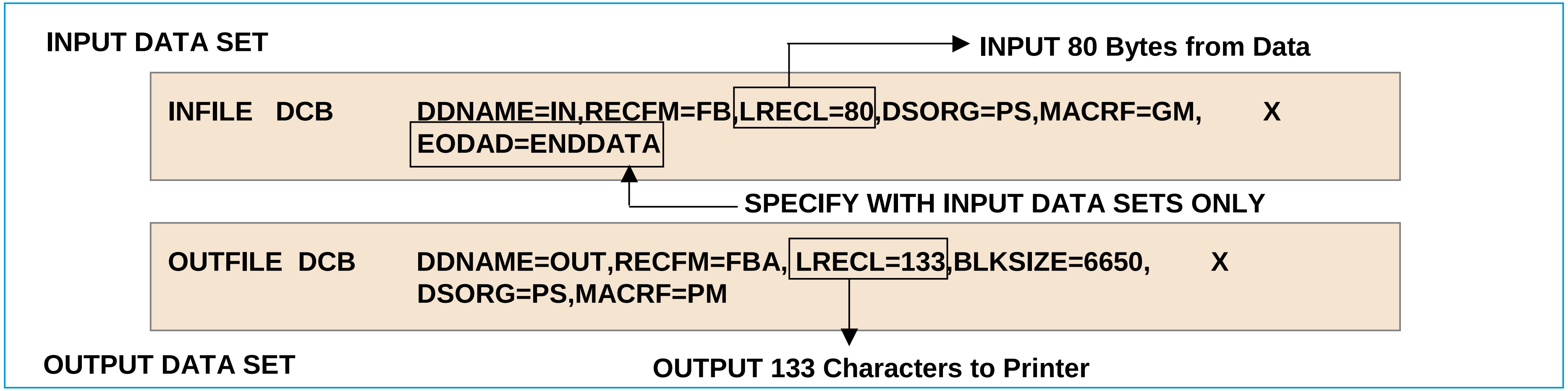
The **MACRF** parameter specifies the format of macros used to access the records in the data set. You specify GM (Move mode of the GET macro) for input, and PM (Move mode of the PUT Macro) for output.

The **RECFM** parameter specifies the record format of the data set being processed. You specify FB (fixed length blocked records) for input, and FBA (fixed length blocked records with American National Standard Institute (ANSI) printer control characters) for output.



Continued…

Kinds of Parameters (cont'd)



INPUT DATA SET
INPUT 80 Bytes from Data
SPECIFY WITH INPUT DATA SETS ONLY
OUTPUT DATA SET
OUTPUT 133 Characters to Printer

```
INFILE   DCB         DDNAME=IN,RECFM=FB,LRECL=80,DSORG=PS,MACRF=GM,        X
                     EODAD=ENDDATA

OUTFILE  DCB         DDNAME=OUT,RECFM=FBA,LRECL=133,BLKSIZE=6650,        X
                     DSORG=PS,MACRF=PM
```

The **LRECL** parameter specifies the length of logical records in the data set. You specify 80 for input and 133 for output.

The **BLKSIZE** parameter specifies the size of physical records, or blocks, in the data set. Omit this parameter for input data sets and use 6650 for output.

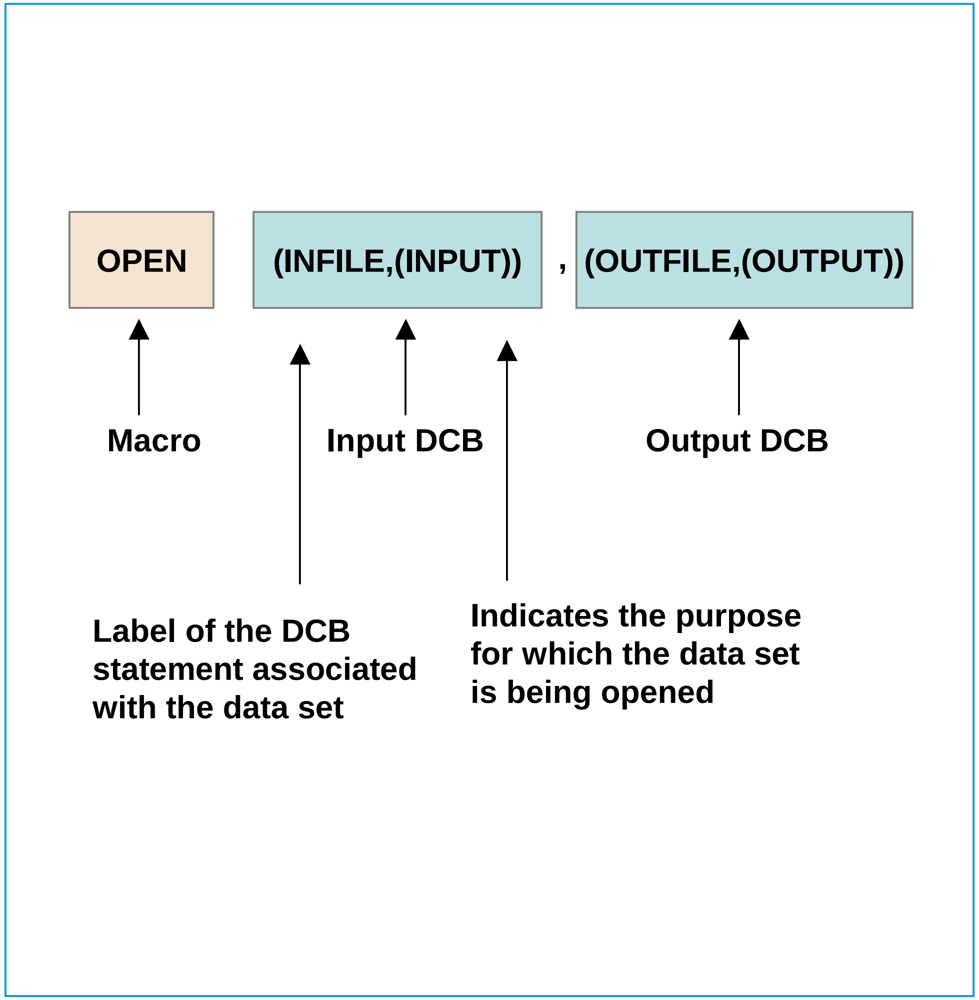
The **EODAD** parameter (End of Data Address) specifies a label in your program where control is to be transferred when a GET is attempted and no more data exists. This parameter is only specified for input data sets.

Defining Data Set Elements

Before input and output operations on data sets can be performed, they must be opened, or made available for processing. You use the OPEN macro instruction for this purpose.

What is the function of the OPEN macro?

The OPEN macro generates executable instructions that invoke operating system routines to check for the presence of a data set, confirm that it matches your specifications, and prepare it for processing. For output data sets, OPEN will create a new data set if the JCL or ALLOCATE statement so specifies.


OPEN
(INFILE,(INPUT))
,
(OUTFILE,(OUTPUT))
Macro
Input DCB
Output DCB
Label of the DCB statement associated with the data set
Indicates the purpose for which the data set is being opened

Defining Data Sets - Summary

Once the DCB has been opened, code data transfer macros, GET for input and PUT for output.

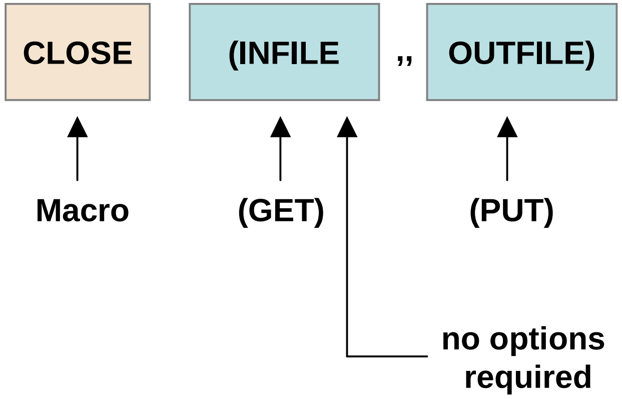
These statements are discussed in more detail in the next section.

Having finished the I/O processing, close the DCBs, prior to ending the program.The CLOSE macro is used for file housekeeping.

The CLOSE macro to close the two data sets is coded as follows:

```
CLOSE (INFILE,,OUTFILE)
```

Close Macro:



Move Mode

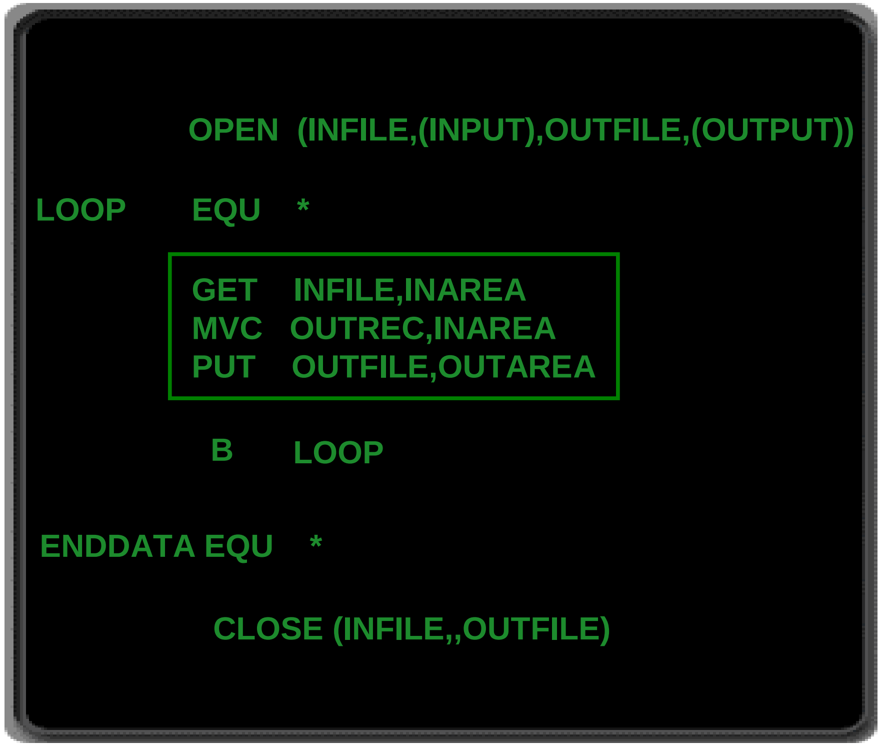
What is Move Mode?

The method by which data is transferred from or to an I/O device is called Move Mode.

Although data is transferred between the I/O device and buffer is main storage, one block at a time, Move Mode I/O moves single logical record to/from a work area. When using this mode, you must define a work area for each data set.

The Move Mode data transfer macro, for input, moves the record from the data set to a buffer in main storage and from the buffer to the work area named INAREA.

The Move Mode data transfer macro, for output, moves the record from the work area named OUTAREA to the buffer in the main storage and from the buffer to the I/O device.



```
         OPEN  (INFILE,(INPUT),OUTFILE,(OUTPUT))
LOOP     EQU   *
         GET   INFILE,INAREA
         MVC   OUTREC,INAREA
         PUT   OUTFILE,OUTAREA
          B    LOOP
ENDDATA  EQU   *
          CLOSE (INFILE,,OUTFILE)
```

Continued…

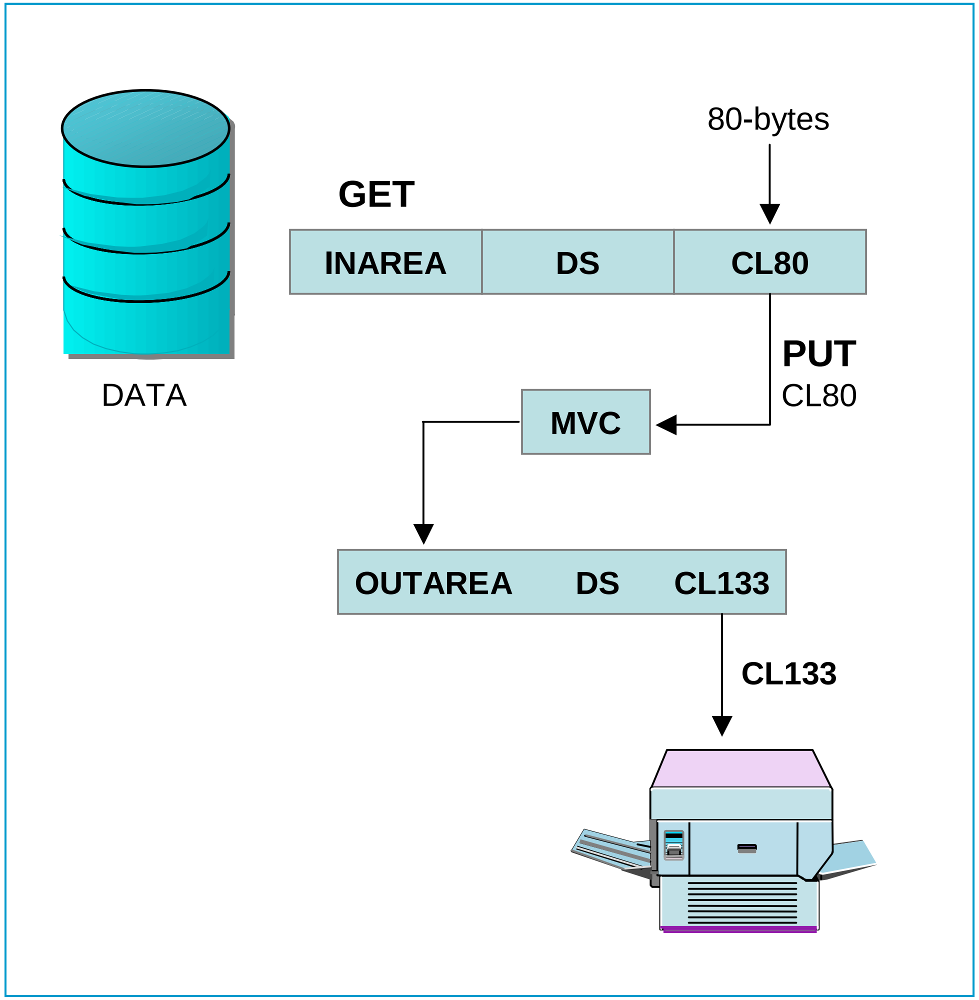
Move Mode (cont'd)

The work area must be the same length as the logical record.

The logical records in the input data set are 80 bytes in length, so the input work area is defined as shown.

The output records are 133 characters in length, so the output work area is defined as shown.

These areas could be subdivided by using the zero duplication factor technique.



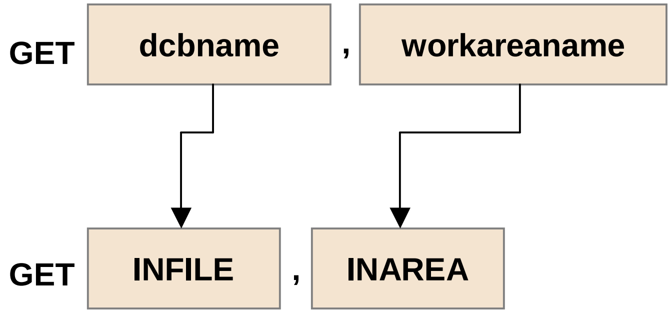
GET Macro

Once a data set is opened, the GET macro reads the next logical record in the data set into a work area. The format of the GET macro in the move mode is shown on the right.

Using the names used up to this point, the macro would be coded as shown on the right.

When the end of the input data is reached, and another GET macro is issued, control is transferred to the address specified as the EODAD in the DCB macro.

Move Mode:



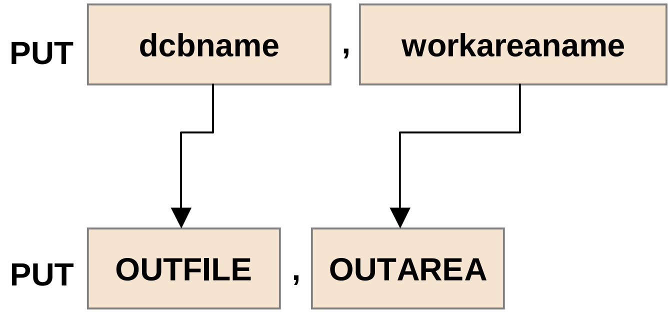
PUT Macro

What is the function of the PUT Macro?

The PUT macro is used to write the next logical record from the work area to the output data set. The format of the PUT macro in move mode is shown on the right.

Using the names used up to this point, coding would appear as shown on the right.

Move Mode:



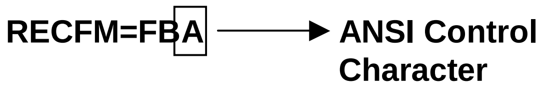
ANSI Control Character

The record format parameter we have used for our output data set is FBA and is coded as shown:

RECFM=FBA

The A specifies that the first byte of each record is an ANSI control character, which is used to specify spacing on the printer.

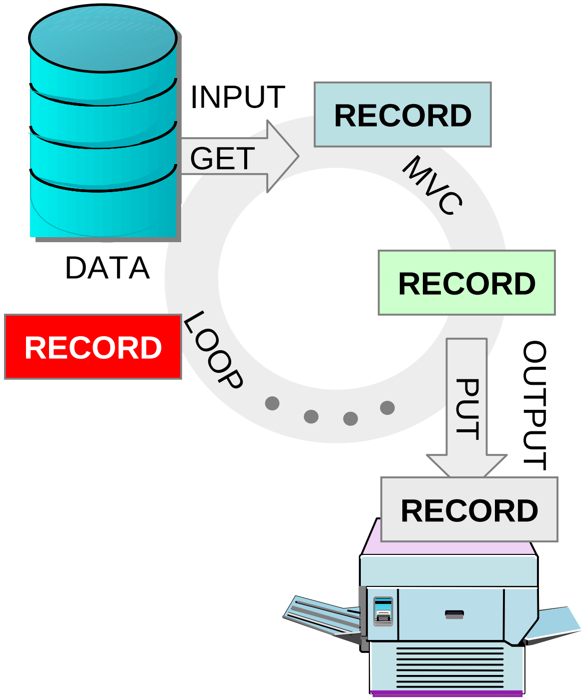
The most common ANSI printer control characters and their meanings are shown on the right.



ANSI Printer Control Character	Meaning
Space	Single space
0	Double space
-	Triple space
+	Suppress space (overprint last line)
1	Skip to top of next page

Consolidating I/O Information

```
LOOP      OPEN   (INFILE, (INPUT) , OUTFILE, (OUTPUT) )   prepare dcbs for processing
          EQU    *
          GET    INFILE, INAREA                           get next input record
          MVC    OUTREC, INAREA                           move to output record
          PUT    OUTFILE, OUTAREA                         write output record
          B      LOOP                                     continue with next record
ENDDATA   EQU    *                                        handle end of data situation
          CLOSE  (INFILE,,OUTFILE)                        close dcbs

INAREA DS CL80
OUTAREA   DS     0CL133                                   specify double spacing
OUTASA DC C'0'                                            copy of input goes here
OUTREC DS CL80                                            make the rest spaces
          DC     CL52 ' '
INFILE    DCB    DDNAME=IN,RECFM=FB,LRECL=80,DSORG=PS,MACRF=GM,    X
                 EODAD=ENDDATA
OUTFILE   DCB    DDNAME=OUT,RECFM=FBA,LRECL=133,BLKSIZE=6650,      X
                 DSORG=PS,MACRF=P
```



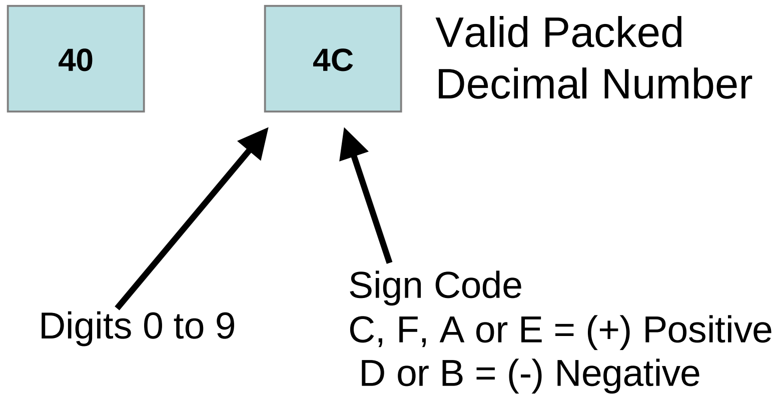
To put all of this I/O information together, you code a program segment to read data from an input data set, and then print it out, double spaced, on the printer.

Packed Decimal Operands

What do packed decimal operands contain?

Packed decimal operands are fields in main storage from 1 to 16 bytes in length. The rightmost byte of a packed decimal field contains a decimal digit in the left nibble and a sign in the right nibble. The preferred signs for packed decimal numbers are C for positive and D for negative. The remaining bytes of a packed decimal number (other than the rightmost) contain two decimal digits, one in each nibble.

Packed Decimal Operand:



Note! If you attempt to perform decimal arithmetic with an operand that is not in valid format, a data exception will occur.

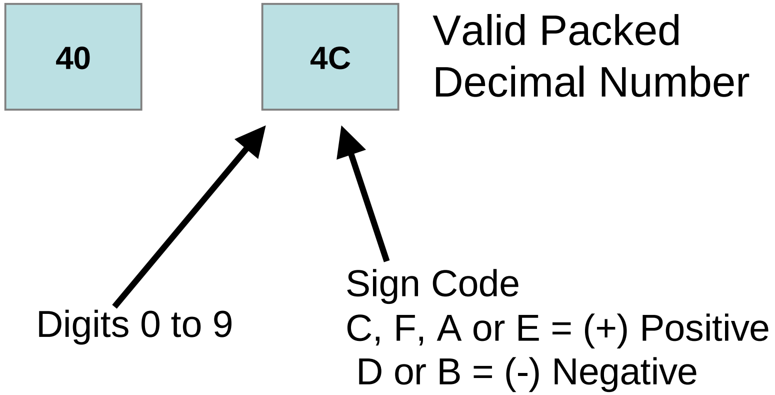
Continued…

Unit: Decimal Arithmetic Topic: Basics of Decimal Arithmetic

Packed Decimal Operands (cont'd)

A packed field n bytes long can hold 2 * n -- 1 digits. The maximum number of digits in a packed decimal number is 31, since the maximum field length is 16. The maximum field length for a multiplier or divisor is 8 bytes.

Packed Decimal Operand:

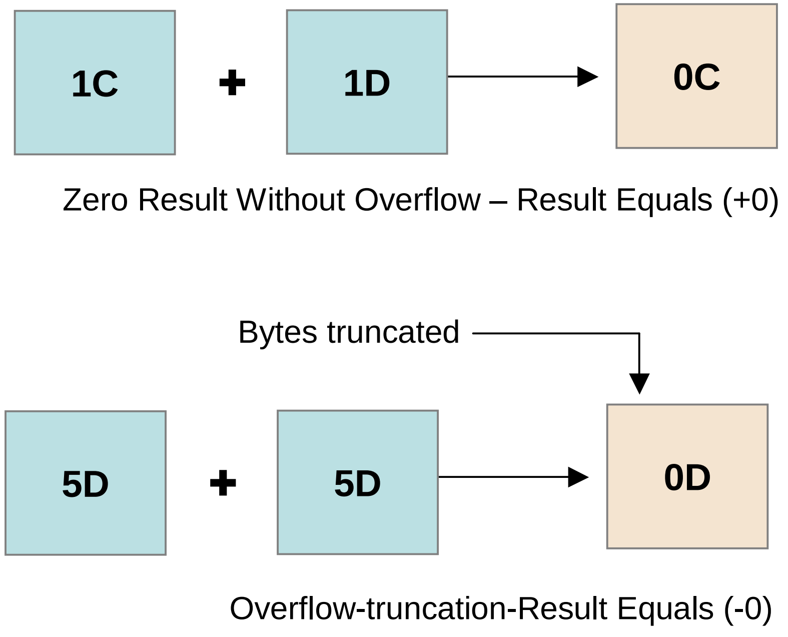


Decimal Arithmetic Instructions

What are decimal arithmetic instructions?

The decimal arithmetic operations are SS (Storage to Storage) (2 length) instructions. Operands do not have to be of the same length. Decimal arithmetic is performed in main storage.

Unlike fixed-point representation, which only provides one zero value, decimal arithmetic allows both +0 and -0 to be represented. If a packed decimal arithmetic operation generates a zero result without overflow, the result will be +0. If overflow occurs, the truncated number is zero, the sign generated is that of the non-truncated number.


1C
+
1D
0C
Zero Result Without Overflow – Result Equals (+0)
Bytes truncated
5D
+
5D
0D
Overflow-truncation-Result Equals (-0)

Condition Code

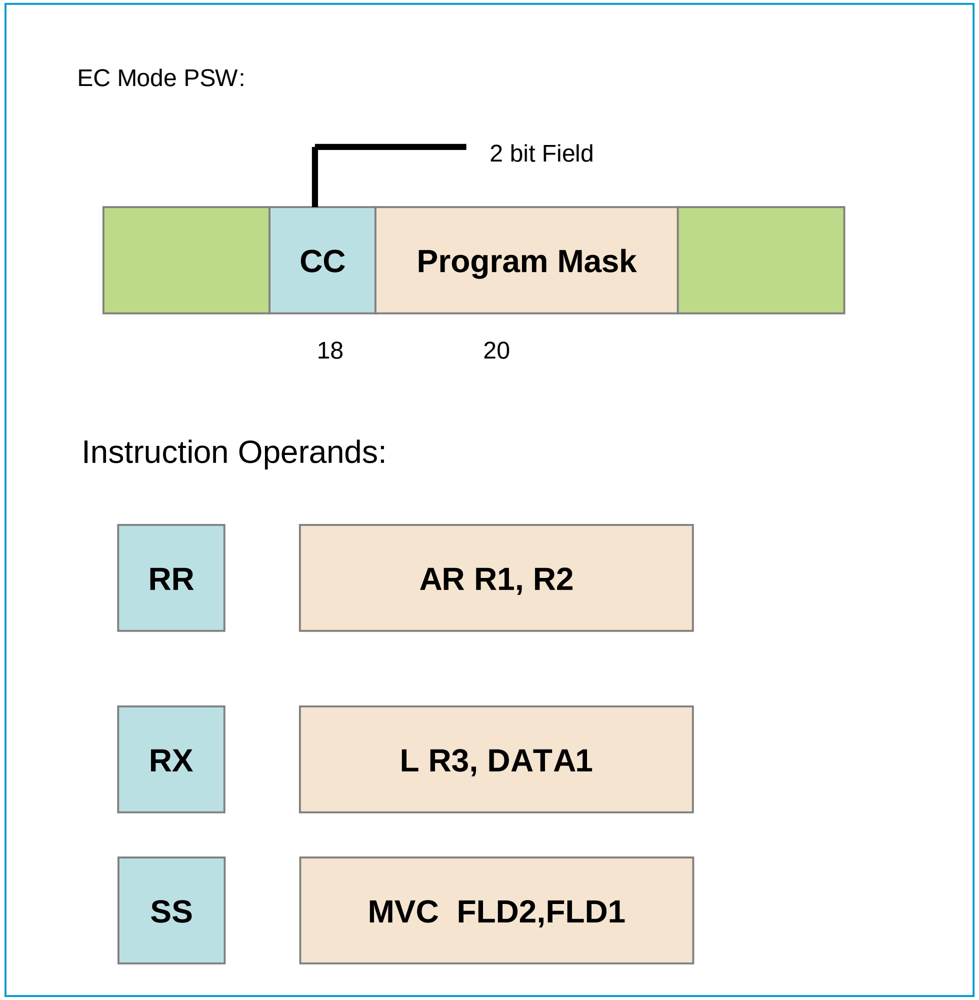
What is a condition code?

The condition code is a two-bit field in the Program Status Word (PSW) that is set by some instructions, and can then be tested by subsequent instructions to implement conditional execution.

Where are instruction operands located?

Instruction operands are located in main storage or in registers. The operand's instruction type determines the location of the operands for each particular instruction:

- RR instructions have two operands in registers
- RX instructions have the first operand in a register and the second in storage.
- SS instructions have both operands in storage.

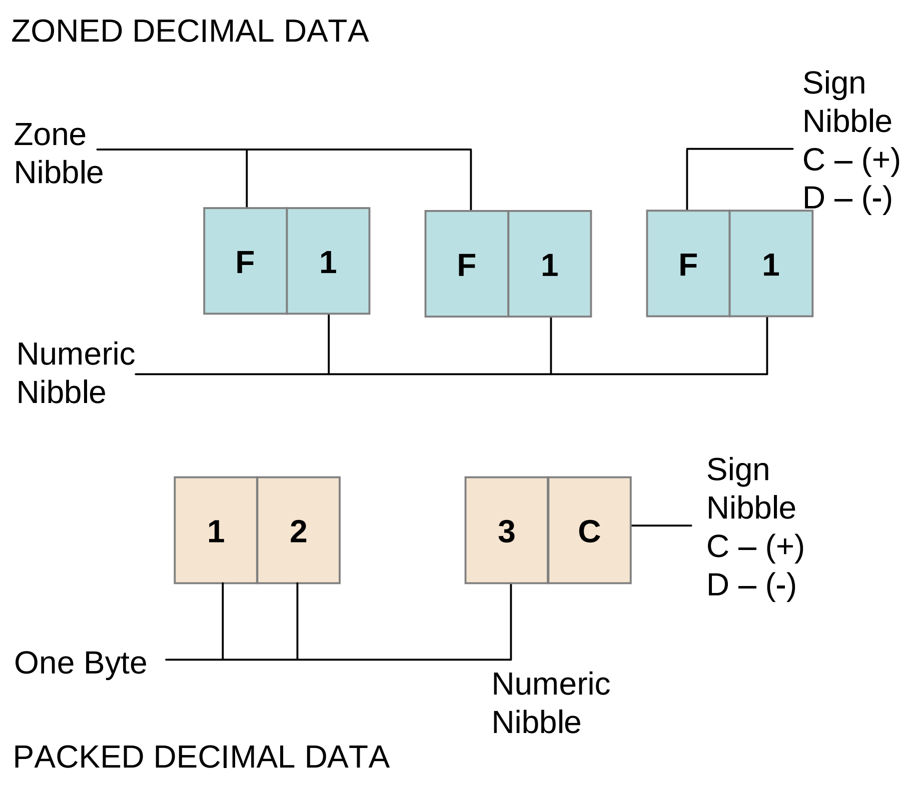


Digits in Zoned and Packed Decimal Types

How many digits do zoned decimal and packed decimal data contain?

Zoned decimal data has one digit per byte, with each byte consisting of a zone nibble and a decimal numeric nibble. The zone in the rightmost byte is the sign.

Packed decimal data consists of two decimal digits per byte for the rightmost, which contains a numeric nibble followed by a sign nibble. Numbers in character form (EDCDIC code) are zoned, with an 'F' sign.



Converting Zoned Decimal Numbers to Packed Decimal Numbers

How do you convert zoned decimal numbers to packed decimal format?

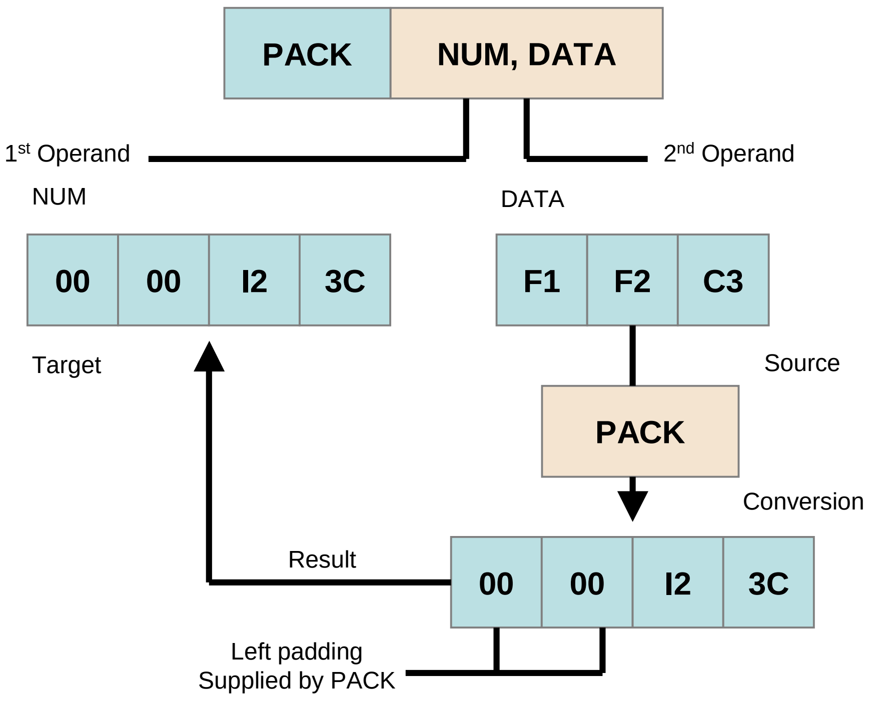
You often need to convert zoned decimal numbers to packed decimal format. The numbers are read in zoned form, but in order to be used in decimal arithmetic they must be packed.

The instruction that performs this conversion is called Pack (PACK) and is shown on the right.

The second operand is converted from zoned to packed format, and the result stored in the first operand.

For example: PACK NUM,DATA

Pack (PACK):
SS Format (2 length) – Storage to Storage



CC setting – Condition Code is Unchanged

Continued…

Converting Zoned Decimal Numbers to Packed Decimal Numbers (cont'd)

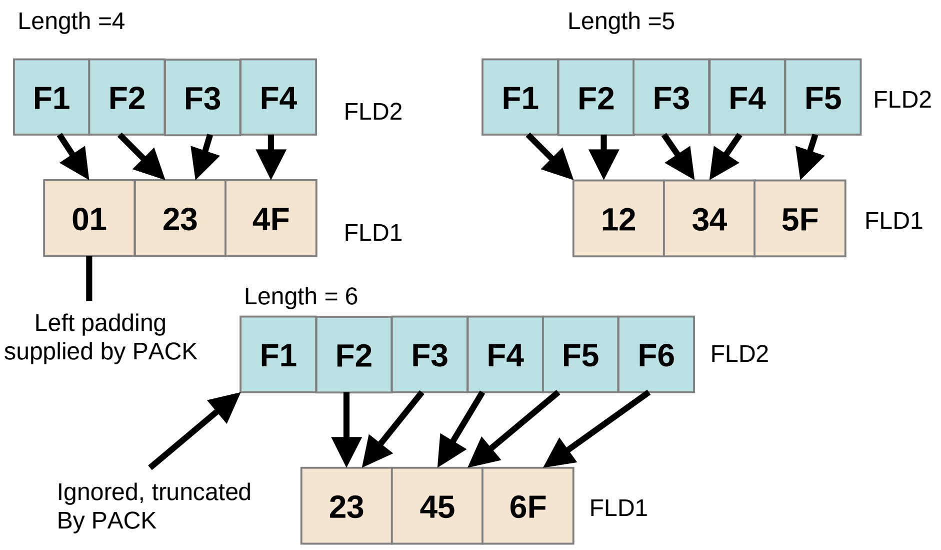
The operation of PACK is as follows:

- It processes each operand right to left. The rightmost byte of the second operand has its nibbles reversed in order and the result is stored in the rightmost byte of the first operand.

- Pairs of bytes are selected from the second operand and combined into a single byte by combining numeric nibbles and discarding zone nibbles. The combined single byte is stored in the first operand.

- If the length of the second operand, L_2, is greater than $2*L_1$-1, left truncation occurs. If L_2 is less than $2*L_1$-1, left padding with zeroes occurs.

Length of FLD2	Contents of FLD2	Resulting value of FLD1
4	F1F2F3F4	01234F
5	F1F2F3F4F5	12345F
6	F1F2F3F4F5F6	23456F



Converting from Packed Decimal to Zoned Decimal Numbers

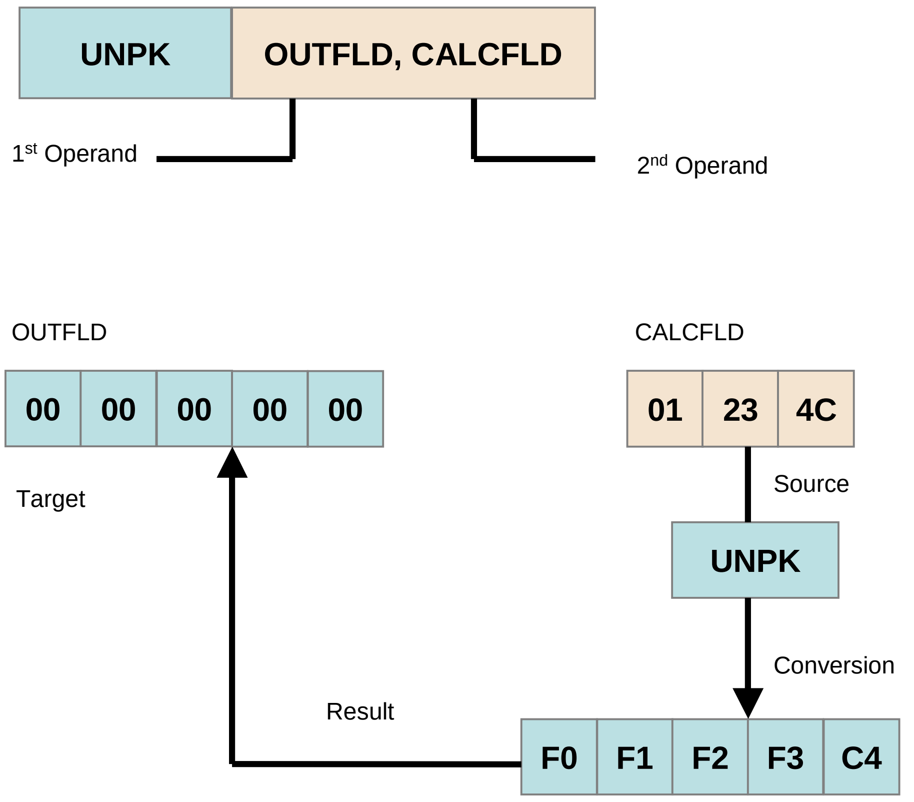
How do you convert from packed decimal to zoned decimal numbers?

The reverse conversion, from packed decimal to zoned decimal is performed by the Unpack (UNPK) instruction.

The second operand is converted from packed to zoned format and the result stored in the first operand.

For example: UNPK OUTFLD, CALCFLD

Unpack (UNPK):
SS Format (2 length) – Storage to Storage



CC setting – Condition Code is Unchanged

Continued…

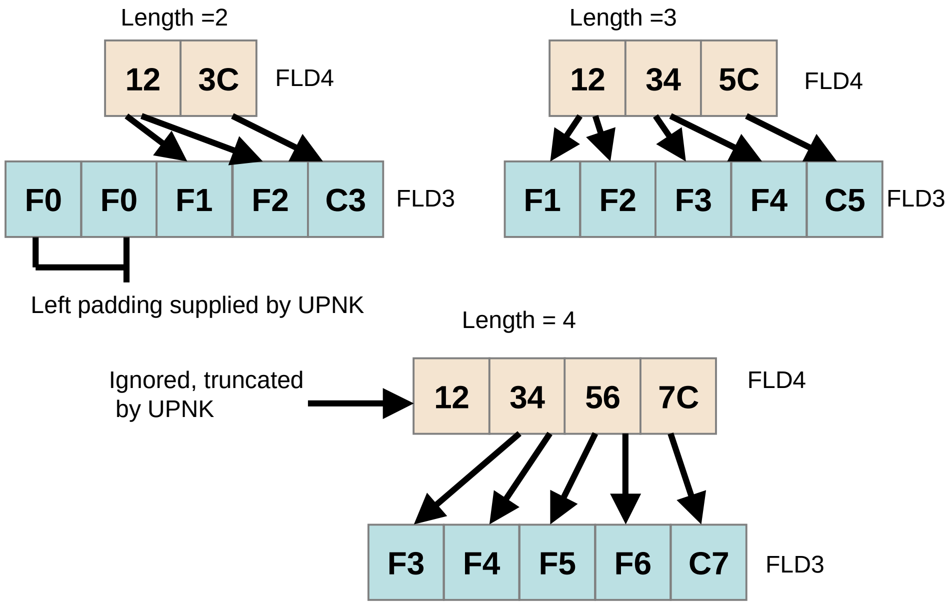
Converting from Packed Decimal to Zoned Decimal Numbers (cont’d)

The operation of UNPK is as follows:

1. It proceeds from right to left through its two operands. The nibbles of the rightmost byte of the second operand are swapped and stored in the rightmost byte of the first operand.
2. Each nibble of the second operand is expanded to a byte by inserting an 'F' in the zone nibble. It is then stored in the first operand.
3. If L_1 is not equal to $2*L_2$-1, left truncation, or left padding with zero will occur.

For example: UNPK FLD3, FLD4

Length Of FLD4	Contents of FLD4	Resulting value Of FLD3
2	123C	F0F0F1F2C3
3	12345C	F1F2F3F4C5
4	1234567C	F3F4F5F6C7



Length =2

12 3C FLD4

F0 F0 F1 F2 C3 FLD3

Left padding supplied by UPNK

Length =3

12 34 5C FLD4

F1 F2 F3 F4 C5 FLD3

Length = 4

Ignored, truncated by UPNK

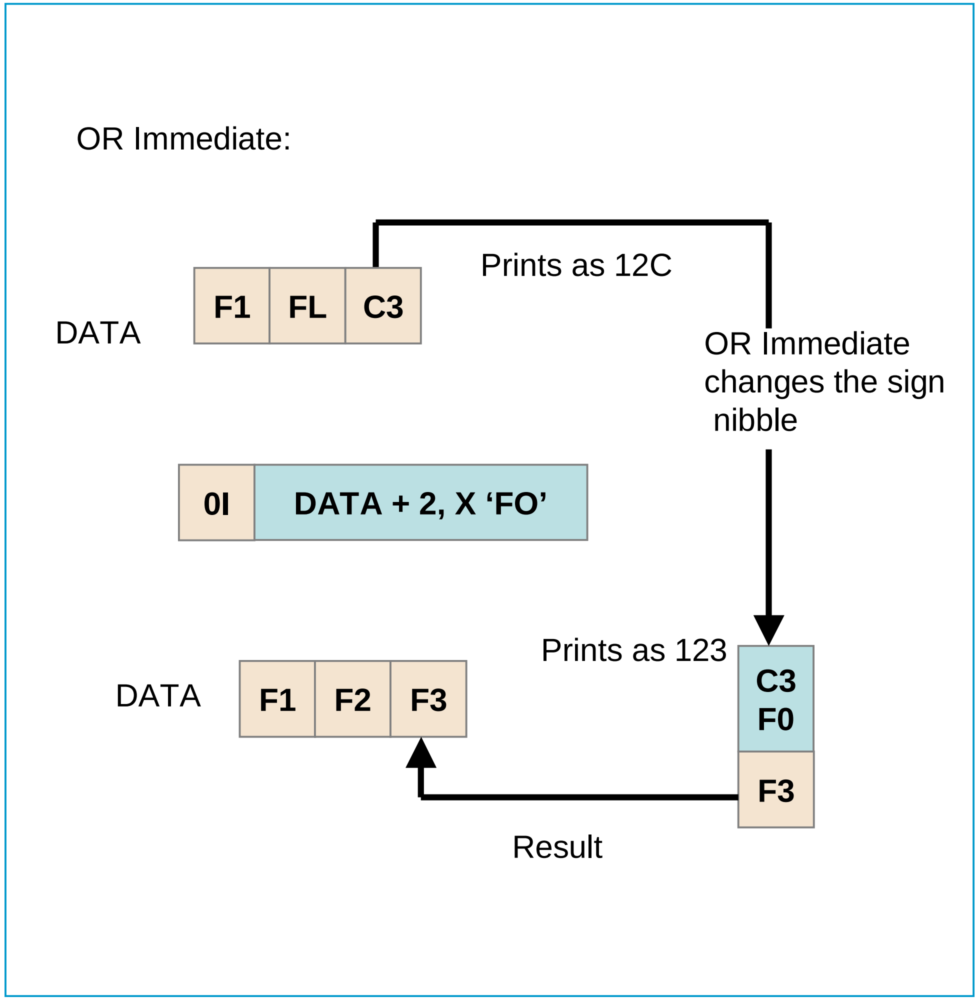
12 34 56 7C FLD4

F3 F4 F5 F6 C7 FLD3

Continued…

Converting from Packed Decimal to Zoned Decimal Numbers (cont'd)

Notice that when you do arithmetic on packed numbers, the signs generated are C for + and D for -. To unpack a number with one of these signs, and interpret the contents as characters, the right most character is not a digit. That is because digits have 'F' in the zone position, while an unpacked number has 'C' or 'D'.

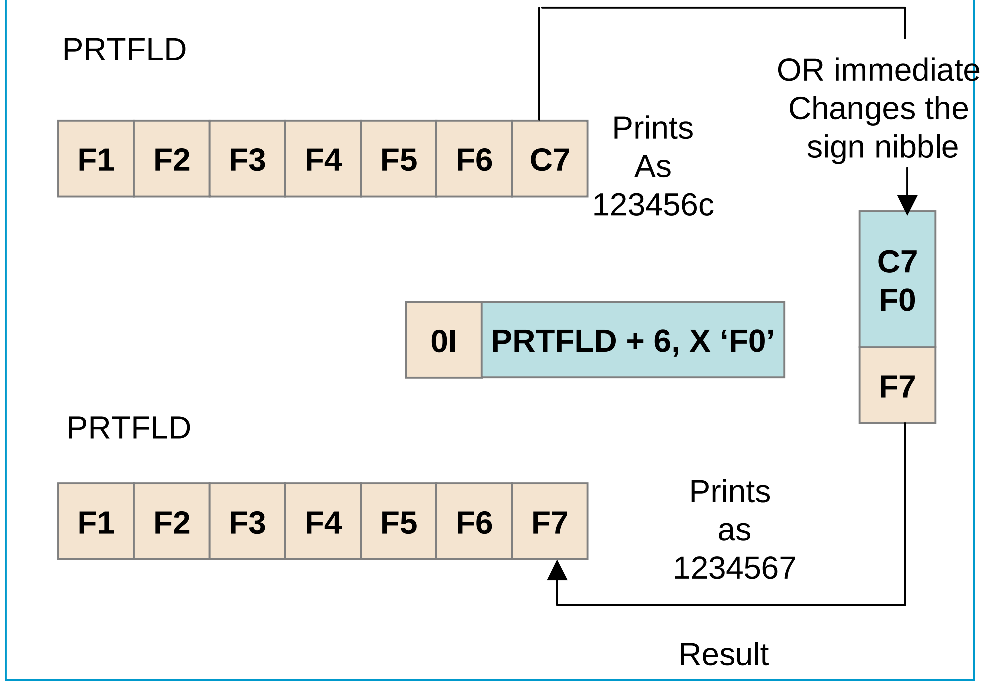


Continued…

Converting from Packed Decimal to Zoned Decimal Numbers (cont’d)

The following code shown on the right formats the contents of a 4 byte packed field called NUM5 for printing.

```
          UNPK   PRTFLD, NUM5
          OI      PRTFLD + 6, X ‘F0’
          .
          .
          .
 NUM5     DS      PL4
PRTFLD    DS      ZL7
```



Zero and Add Instruction

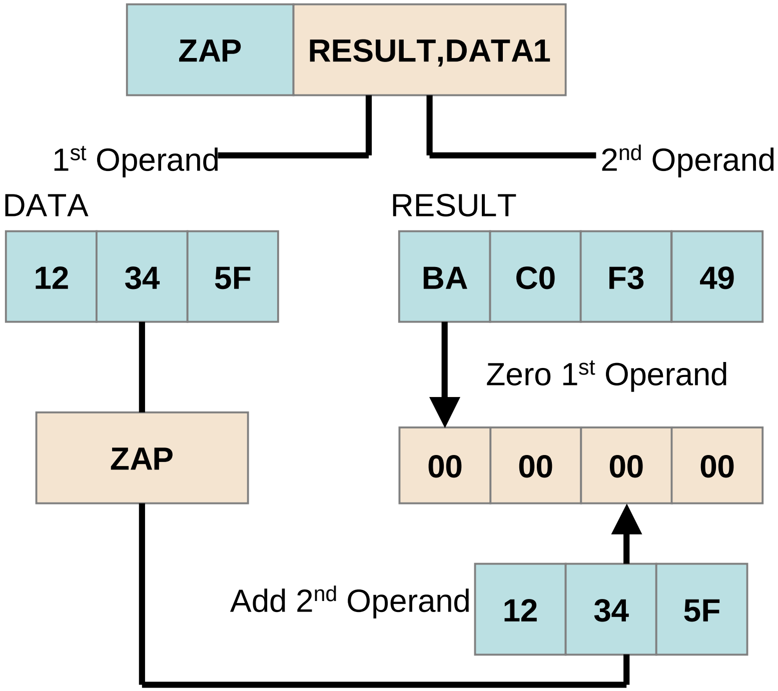
What is the Zero and Add Instruction?

The Zero and Add (ZAP) instruction is used to move packed data from one main storage location to another.

The field sizes of the sending and receiving fields do not have to be equal. However, overflow can occur if the receiving field is not long enough to hold all of the significant digits of the sending field. The sending field must contain a valid packed number , but the initial contents of the receiving field are not checked. The operation proceeds as if the receiving field was zeroed, and then the spending field is added to it.

For example: ZAP RESULT, DATA1

Zero and Add (ZAP):
SS Format (2 length) – Storage to Storage



CC setting: 0 – result is zero
1 - result is negative
2 - result is positive
3 - overflow

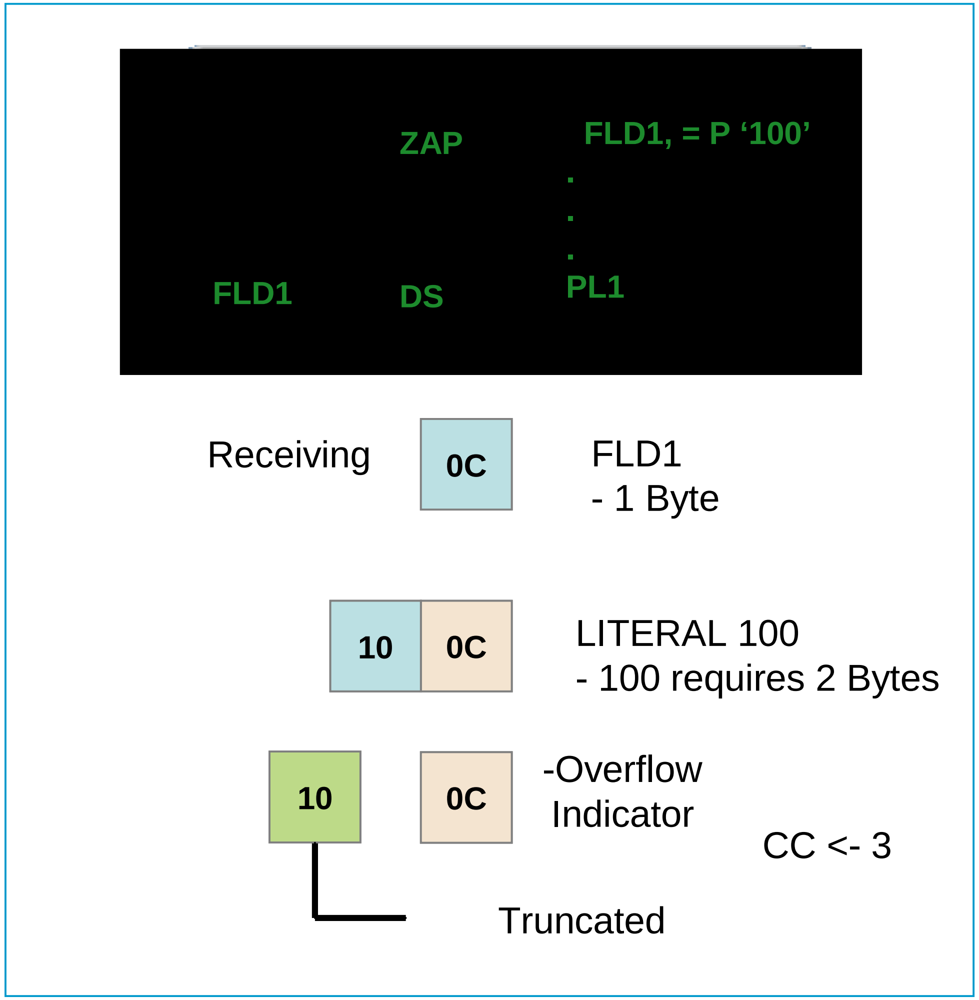
Continued…

Zero and Add Instruction (cont'd)

The condition codes shown previously are standard for many arithmetic instructions. The abbreviation "CC setting: Arithmetic" will be in reference to these condition code settings.

The overflow condition occurs when the magnitude of an arithmetic result is too large to fit into the result field.

You would get an overflow condition if you executed the instruction on the right. 100 will not fit into a 1 byte packed field. 10 is truncated and CC is set to 3.



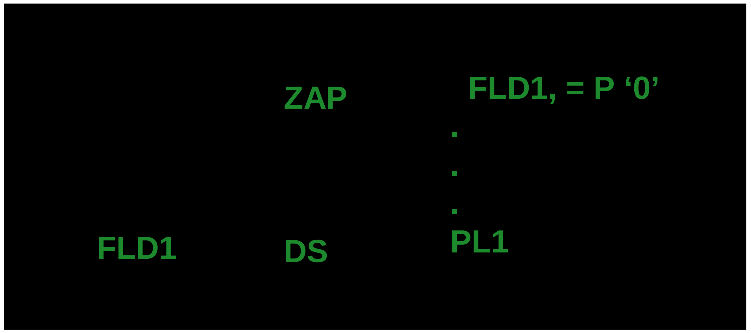
```
            ZAP     FLD1, = P '100'
                    .
                    .
                    .
    FLD1    DS      PL1
```

Continued…

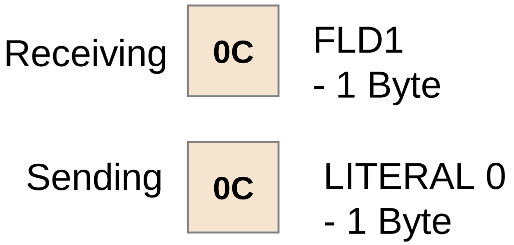
Zero and Add Instruction (cont’d)

In the latter case, the program may seem to execute correctly but produce incorrect numeric results. It is important to give sufficient thought to choosing appropriate field lengths and to use greater lengths if you are unsure.

For example: ZAP FLD1,=P'0'



```
          ZAP     FLD1, = P ‘0’
                  .
                  .
                  .
FLD1      DS      PL1
```



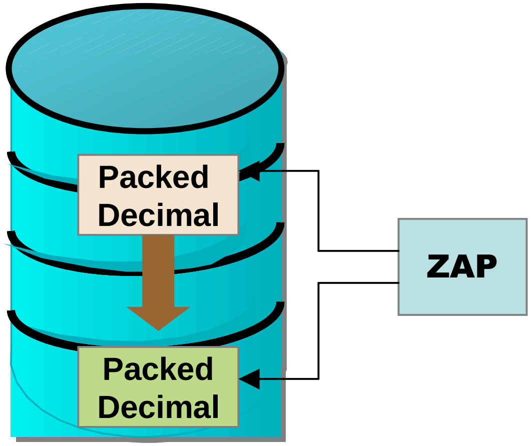
The fields FIT
CC - 0

Continued…

Zero and Add Instruction (cont'd)

ZAP is used primarily to initialize packed decimal fields and to move packed data from one location to another.

For some operations, like multiplication and division, it is necessary to place an operand in a larger field for the operation to work successfully. ZAP is used in these cases.



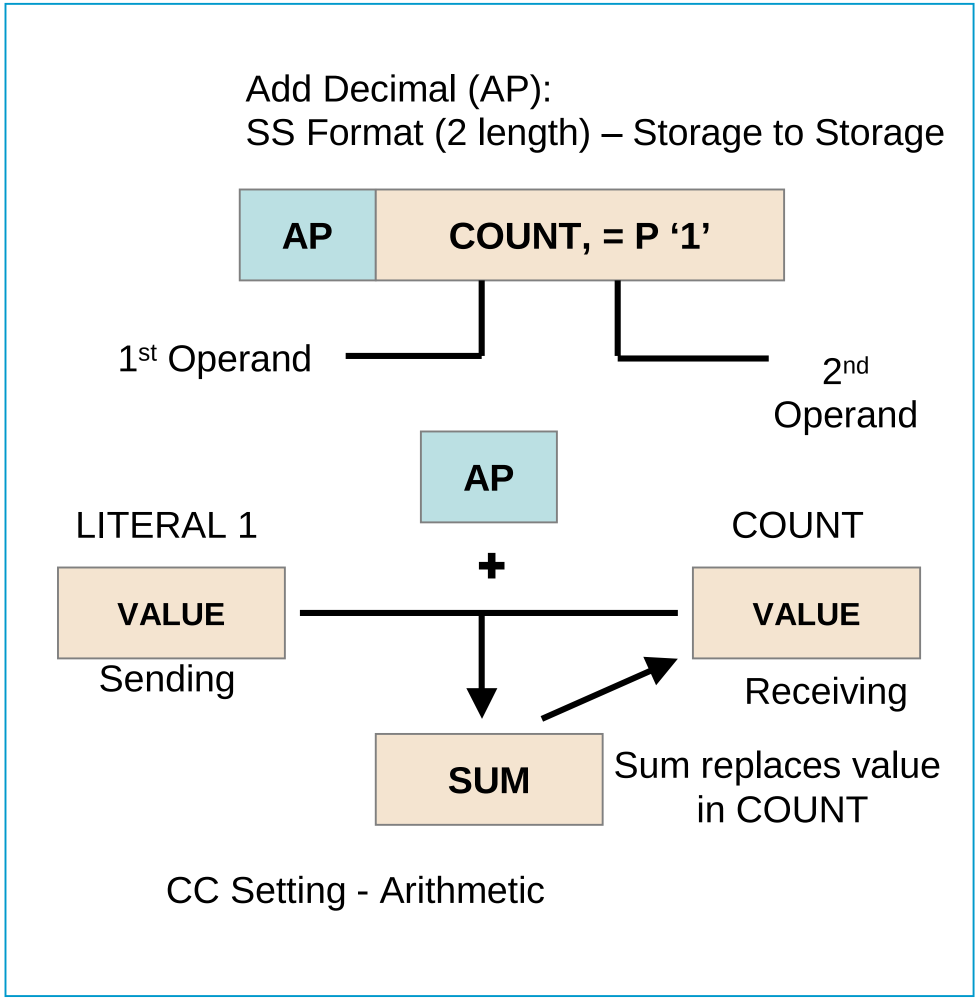
Add Decimal

What is the function of Add Decimal?

The Add Decimal (AP) adds a packed decimal sending field (second operand) to a packed decimal receiving field (first operand). It places the sum in their receiving field, and sets the condition code according to the result.

For example: AP COUNT,=P'1‘

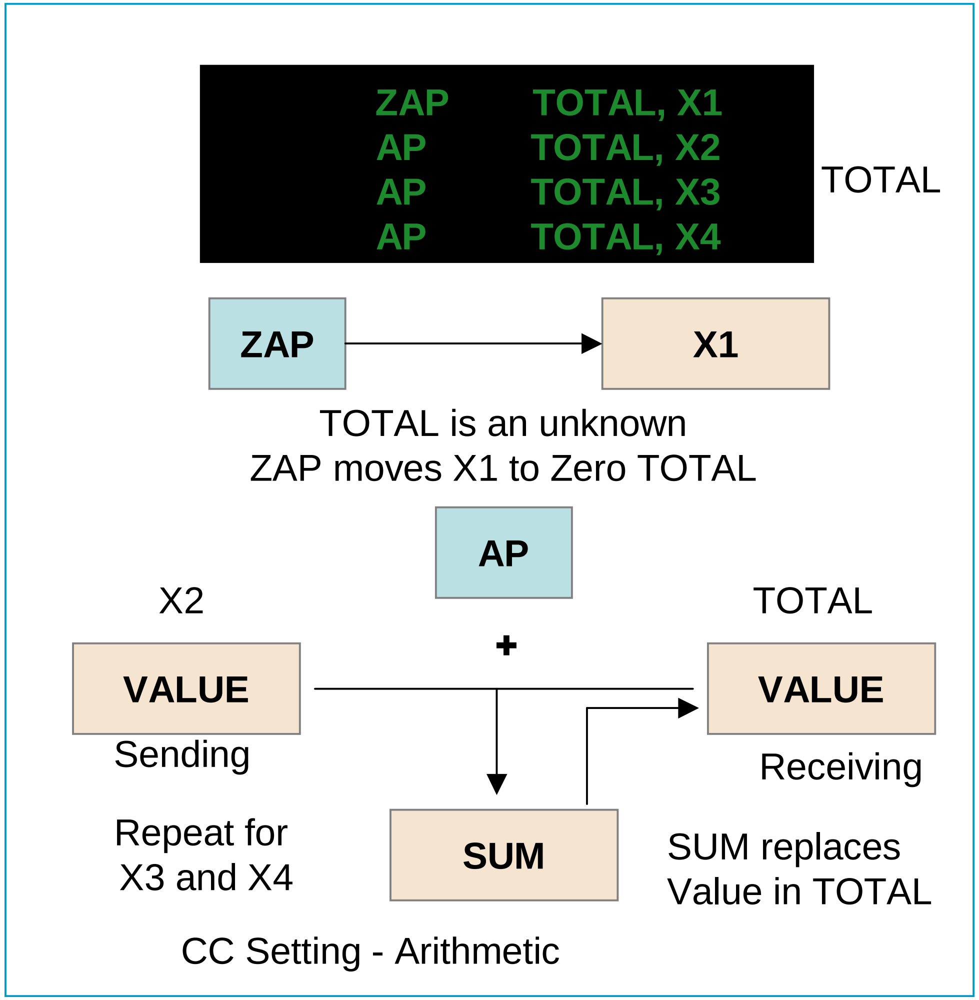
A literal value of one is added to count.



Continued…

Add Decimal (cont’d)

To sum 4 values called X1, X2, X3 and X4 into a field called TOTAL, you could use the code shown on the right.



```
ZAP    TOTAL, X1
AP     TOTAL, X2
AP     TOTAL, X3
AP     TOTAL, X4
```

Note! The first instruction has to be a ZAP, since no information is given about whether TOTAL was initialized to zero.

Subtract Decimal

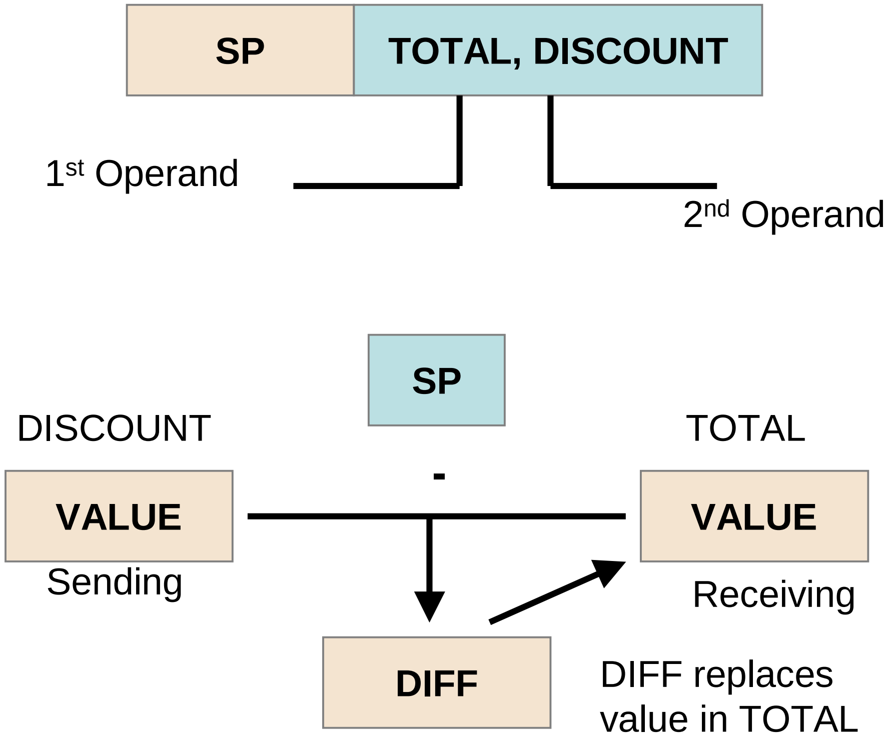
What is the function of the Subtract Decimal?

The Subtract Decimal (SP) instruction subtracts a packed decimal in the sending field (second operand) from a packed decimal in the receiving field (first operand), places the difference in the receiving field (first operand), and then sets the condition code according to the results.

For example: SP TOTAL, DISCOUNT

The contents of DISCOUNT are subtracted from TOTAL, and the difference placed in TOTAL.

Subtract Decimal (SP):
SS Format (2 length) Storage to Storage



DIFF replaces value in TOTAL

CC Setting - Arithmetic

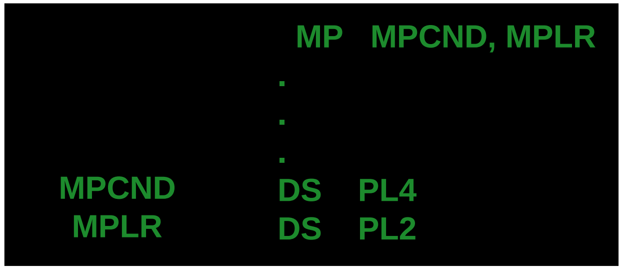
Multiply Decimal

What are the functions of the multiplier and multiplicand?

Decimal multiplication takes into account the fact that the length of the product can be as long as the combined lengths of multiplier and multiplicand.

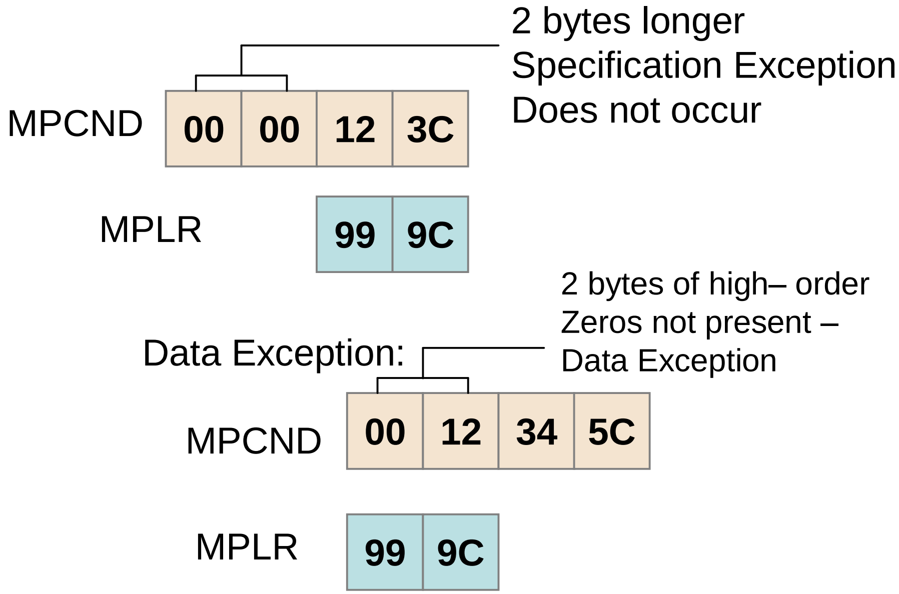
The length of the multiplier may not exceed 8 bytes, and must be less than the length of the multiplicand, which also holds the product. If these conditions are not satisfied, a specification exception occurs.

Also, when the instruction is executed the multiplicand must have as many bytes of high order zeros, as there are bytes in multiplier. If this condition is not met, a data exception occurs.



```
                MP   MPCND, MPLR
                .
                .
                .
MPCND           DS   PL4
MPLR            DS   PL2
```

Specification Exception:



Signs

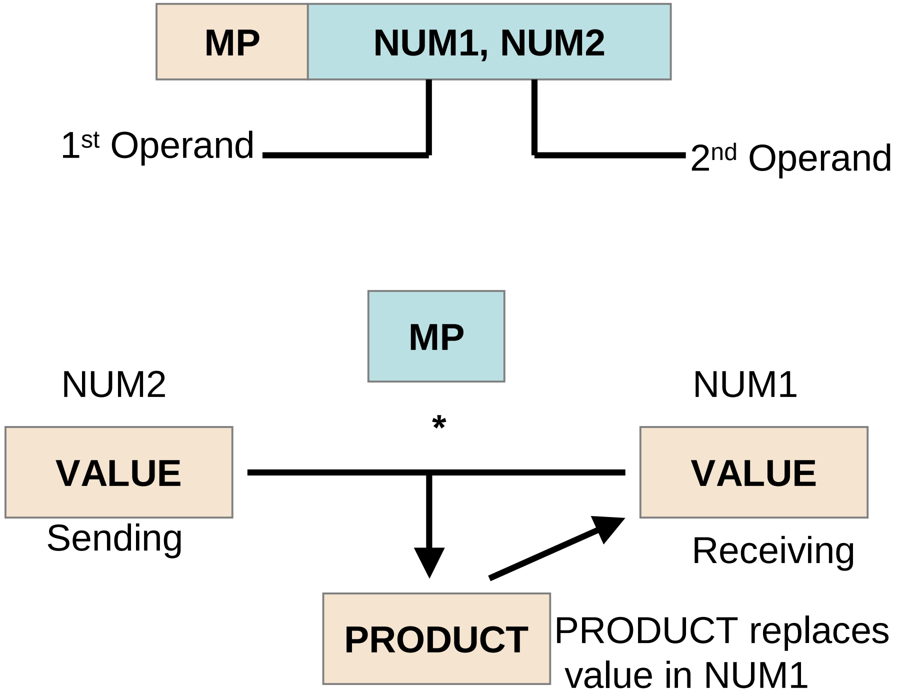
How do you determine the sign of a product?

The sign of the product is determined by the rules of algebra. That is, if the signs of the multiplier and the multiplicand are the same, the product is positive, otherwise the product is negative. This rule holds even when either multiplier or multiplicand is zero, so a negative zero result can occur.

The Multiply Decimal (MP) instruction multiplies a packed decimal multiplicand contained in the first operand storage location by a packed decimal multiplier contained in the second operand storage location.

For example: MP NUM1, NUM2

Multiply Decimal (MP):
SS Format (2 length) – Storage to Storage



CC Setting – Condition Code is Unchanged

Field Lengths

How do you avoid problems with field lengths?

A standard technique to avoid problems with field lengths using Multiply Decimal is to use a field which is the sum of the lengths of multiplier and the multiplicand for the product. If NUM1 was a 5 byte field, and NUM2 a 3 byte field, you would define a field 8 bytes long to hold the product. The code to perform the multiplication is shown on the right.

```
                  ZAP  PRODUCT.NUM1
                  MP   PRODUCT.NUM2
                  .
                  .
                  .
NUM1              DS       PL5
NUM2              DS       PL3
PRODUCT           DS       PL8
```

Length of NUM1		Length of NUM2		Length of Product
5 Byte	+	**3 Byte**	=	**8 Byte**

To ensure that you have enough Low – Order Zeros, the Product field must be > or =to the Sum of the lengths of Multiplicand and Multiplier.

Divide Decimal

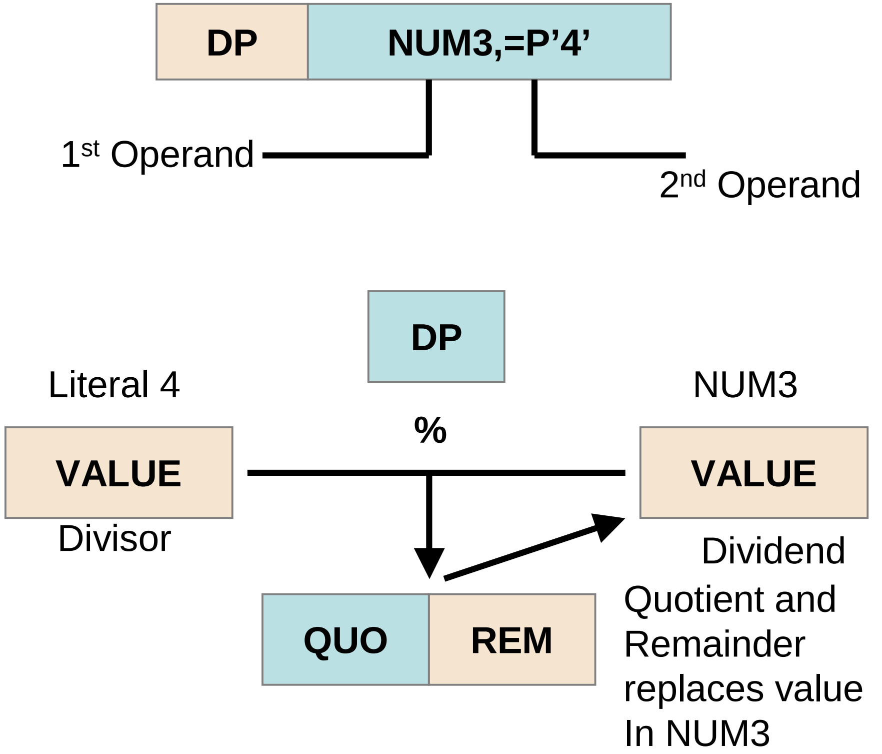
What is the function of the Divide Decimal?

The Divide Decimal (DP) instruction divides a packed decimal dividend (first operand) by a packed decimal divisor (second operand). The quotient and the remainder replace the dividend in the first operand.

For example: DP NUM3,=P'4'

NUM3 is divided by 4, with the quotient replacing all but the rightmost byte of NUM3, and the remainder in the rightmost byte.

Divide Decimal (DP):
SS Format (2 length) – Storage to Storage



CC Setting – Condition Code is Unchanged

Continued…

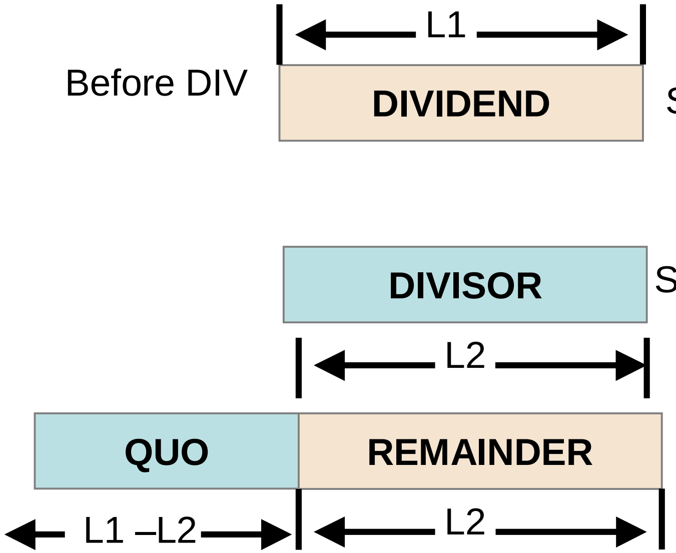
Divide Decimal (cont'd)

Decimal division is integer division and so it produces both an integer quotient and an integer remainder. After the division, the quotient and remainder share the first operand field that initially contained the dividend.

Both quotient and remainder are packed decimal fields, with the quotient preceding the remainder. The length of the quotient field is equal to the length of the dividend field, minus the length of the divisor field. The length of the remainder field equals the length of the divisor field.

A specification exception results if the dividend field is not longer than the divisor field, or if the divisor length is greater than 8 bytes. If the divisor is zero, or the quotient is too large to fit into the quotient field, a decimal divide exception occurs.

Decimal Division:



L1 = Length of Dividend in Bytes
L2 = Length of Divisor in Bytes
Rules:
L1 – Maximum of 16 Bytes
L2 – Maximum of 8 Bytes
L1 must be greater than L2

Continued…

Divide Decimal (cont’d)

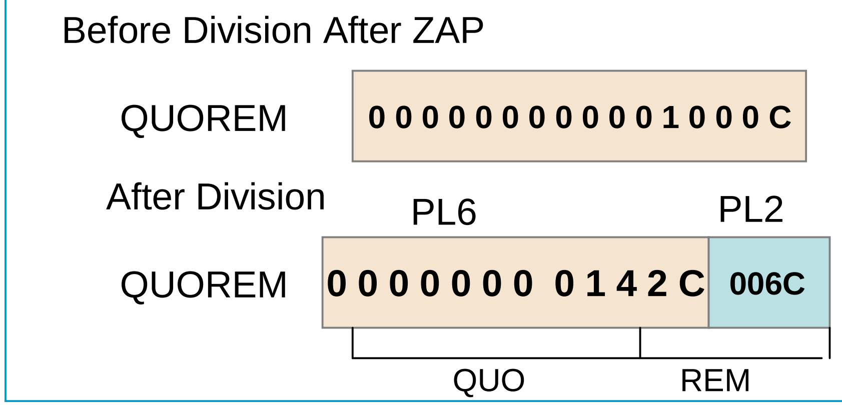
A useful technique in performing decimal division is to move the dividend to a field as large as the sum of the dividend and the divisor field lengths, prior to the division. You can use the technique of zero duplication factor to define this single field in two ways: one for the dividend (before the division), and one for the quotient and remainder (after the division).

To divide NUM4 (6 bytes long) by NUM5 (2 bytes long), the code shown on the right could be used.

After the DP instruction, the quotient is in QUOTIENT and the remainder in REMAINDER.

```
                 ZAP  QUOREM,NUM4
                 DP   QUOREM,NUM5
                 .
                 .
                 .
NUM4             DS   PL6
NUM5             DS   PL2
QUOREM           DS   PL8
QUOTIENT         DS   PL6
REMAINDER        DS   PL2
```

Assume **NUM4 = 1000**
NUM5 = 7



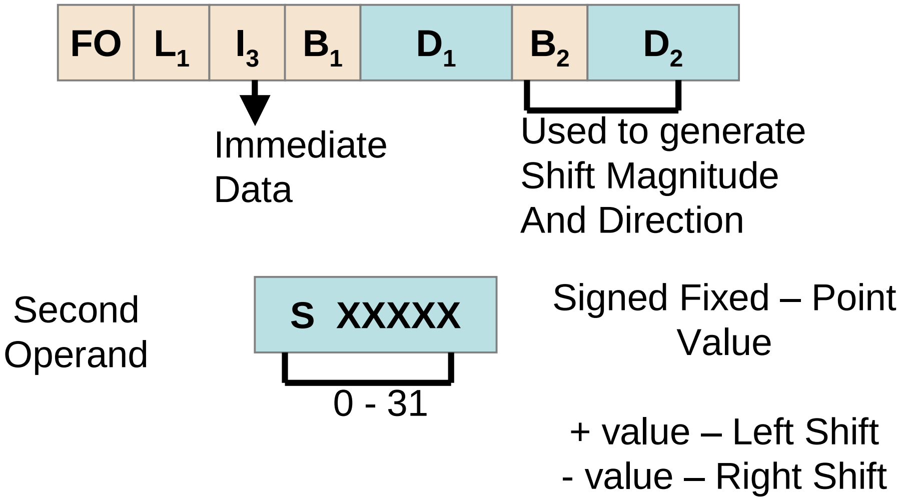
Shift and Round Decimal Instruction

What is the function of the Shift and Round Decimal Instruction?

The Shift and Round Decimal Instruction (SRP) allows you to multiply and divide packed decimal numbers by powers of ten, and to round when right shifting (dividing).

The SRP instruction is a two length SS format instruction, but with a difference. The first operand specifies the number to be shifted. What would normally be the L_2 field is called I_3 and holds the rounding factor.

Shift and Round Decimal (SRP):
SS Format (2 length) – Storage to Storage



FO	L_1	I_3	B_1	D_1	B_2	D_2

Immediate Data

Used to generate Shift Magnitude And Direction

Second Operand

S XXXXX

0 - 31

Signed Fixed – Point Value

+ value – Left Shift
- value – Right Shift

CC Setting - Arithmetic

Continued…

Unit: Decimal Arithmetic Topic: Decimal Arithmetic Operations

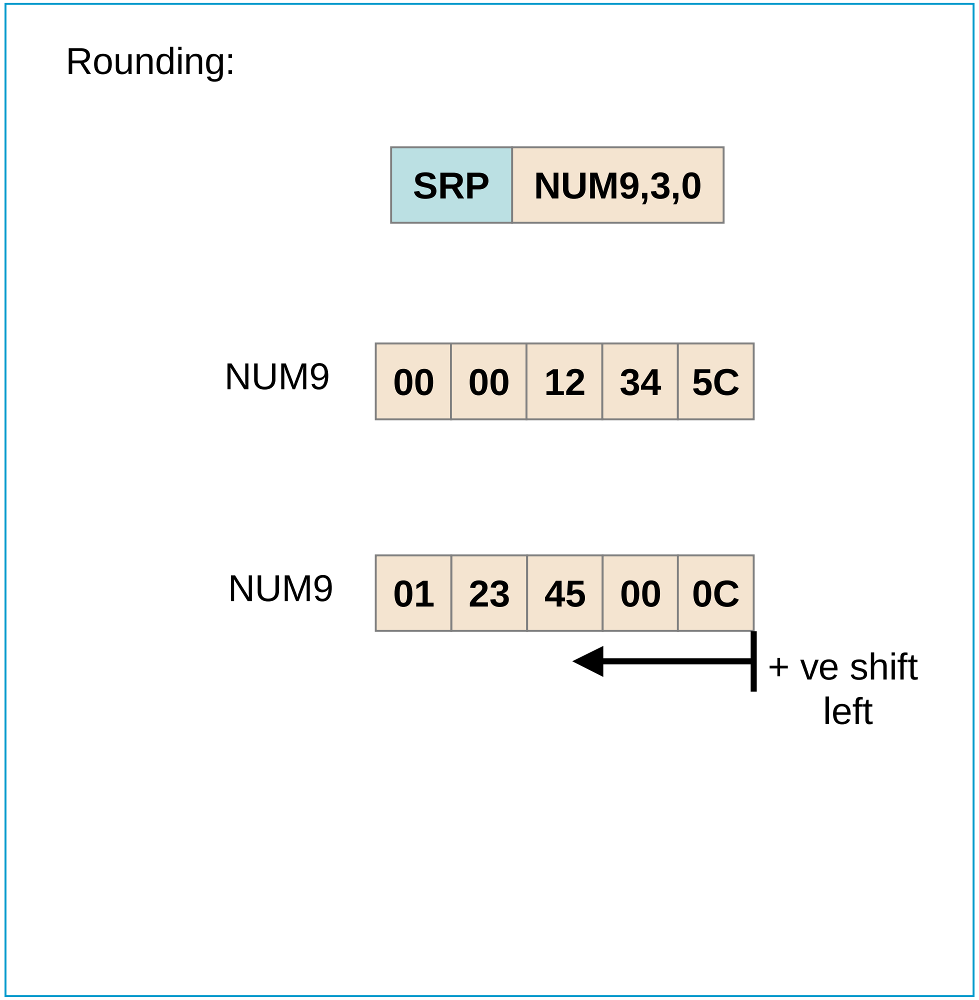
Shift and Round Decimal Instruction (cont'd)

If a significant digit is shifted out during a left shift, the condition code is set to indicate overflow.

The first operand is shifted the number of positions and direction specified by the second operand. In the case of a right shift, the absolute value of the first operand is rounded by the rounding factor.

For example: SRP NUM9,3,0

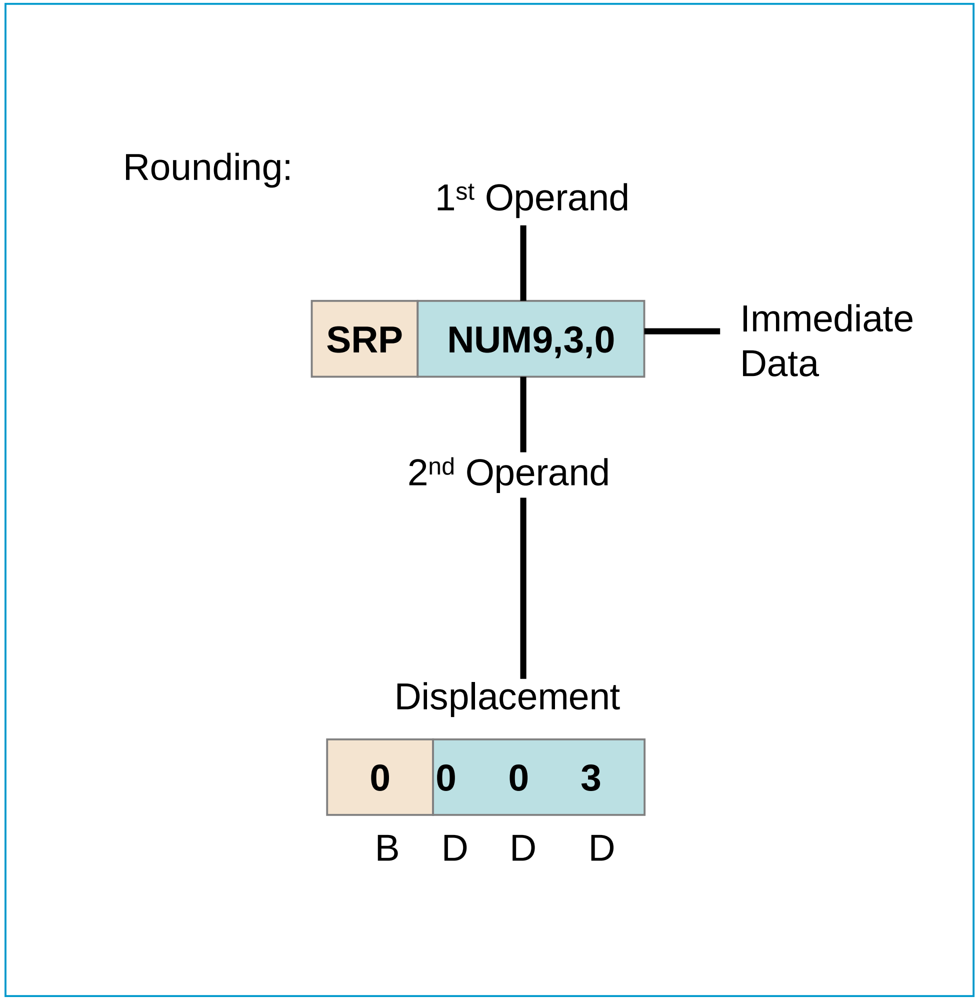
NUM9 is shifted 3 decimal positions to the left. This is equivalent to multiplying NUM9 by 1000.



Continued…

Shift and Round Decimal Instruction (cont'd)

It is useful to look at the syntax of the above instruction before proceeding. NUM9 is the first operand. 3, the second operand, is a displacement field only. 0 is the immediate operand, the rounding factor.



Continued…

Shift and Round Decimal Instruction (cont'd)

If you want to specify a right shift of 3, you might be tempted to specify -3 as the second operand. This would be incorrect. Displacements must be in the range of 0-4095. Negative displacements are invalid. What is needed in the displacement field is the value that corresponds to the 6 bit value -3, in sign and two's complement form. Calculate this as 111101_2, which is 61_{10}. To shift NUM9 3 positions to the right, with half rounding, specify this as follows.

```
SRP   NUM9,61,5
```

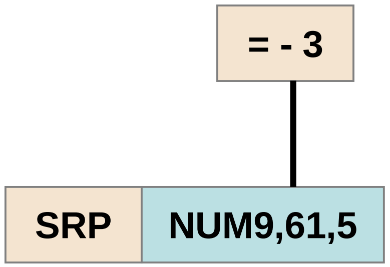
Rounding:

00011 = +3

11100 – Flip Bits

+1 – Add 1

11101 = -3 Twos Complement

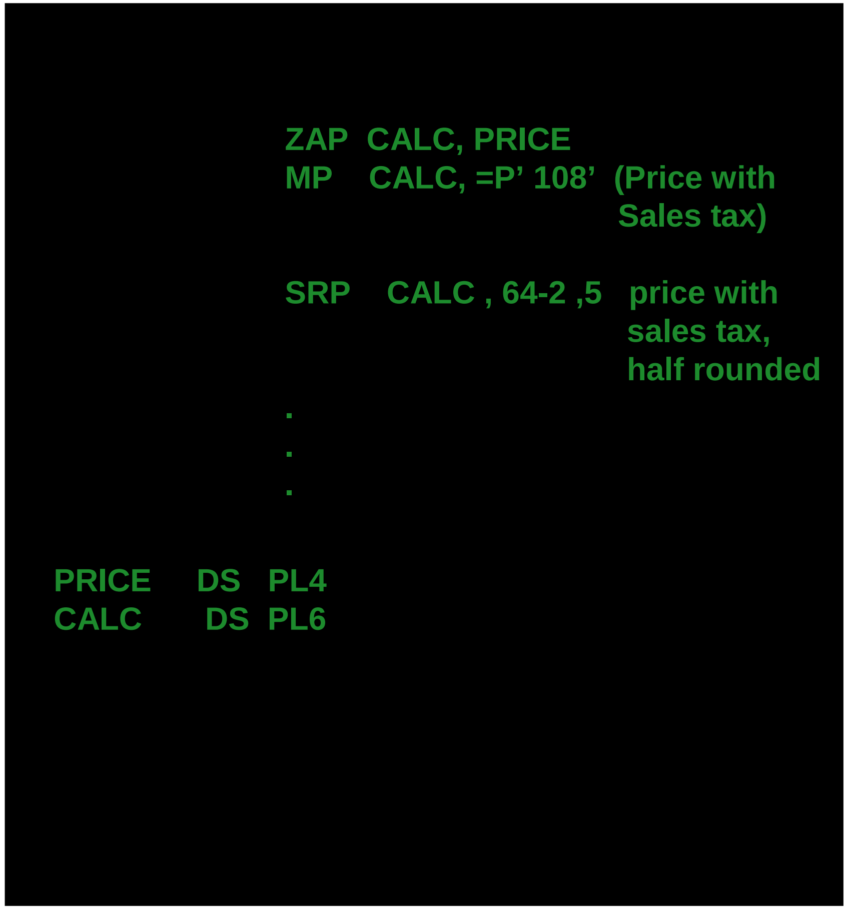


Continued…

Shift and Round Decimal Instruction (cont’d)

In general, to shift n positions to the right, specify the second operand as 64-n. You can actually code the second operand of SRP as follows, to make the code more clear.

SRP NUM9,64-3,5 shift right 3 positions and half round

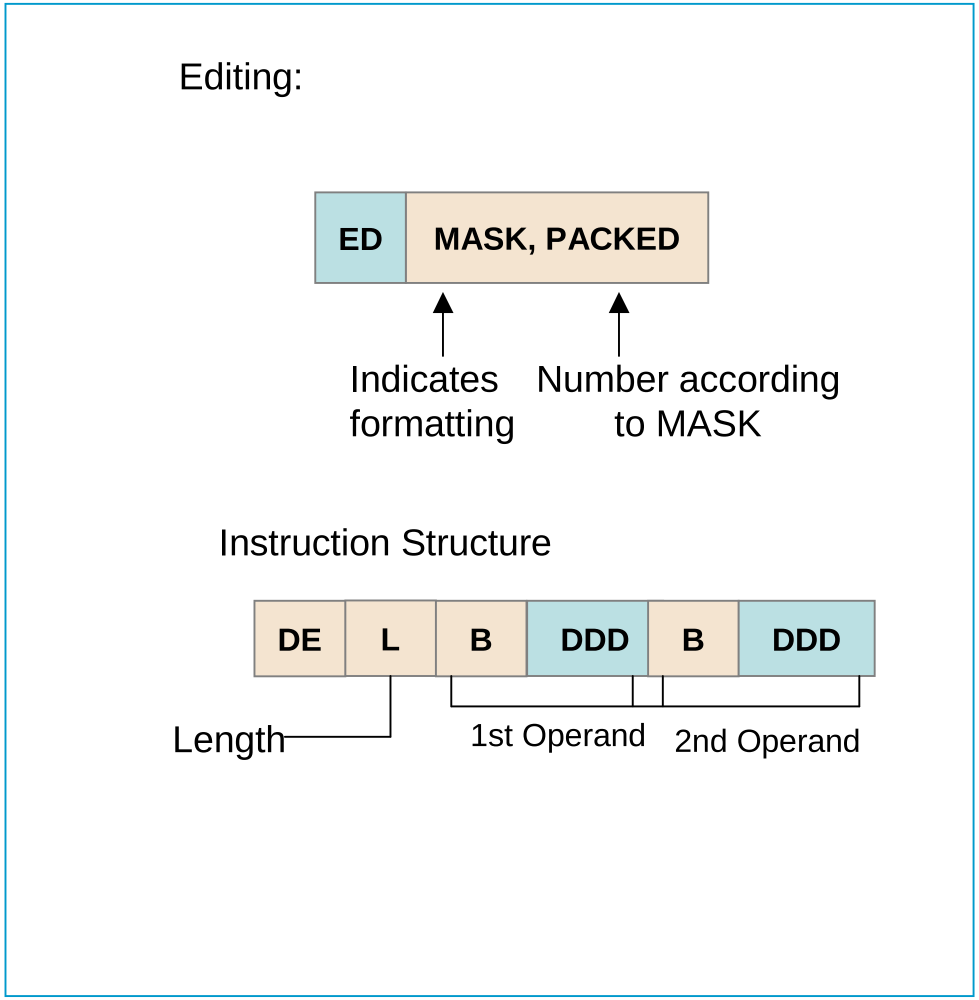


```
              ZAP  CALC, PRICE
              MP   CALC, =P’ 108’  (Price with
                                    Sales tax)

              SRP   CALC , 64-2 ,5  price with
                                    sales tax,
                                    half rounded
              .
              .
              .


PRICE     DS  PL4
CALC       DS PL6
```

Editing Decimal Data

For many programs, it is important to present numeric results in a format that is easy to read and interpret. If you are dealing with large numbers, you can use commas to break the number up for readability . Numbers such as 123,456,789 are easier to read and interpret than 123456789.

If you are doing currency calculations, you might want to present your results with a dollar sign, and with a period separating dollars and cents. In this case, your output might be presented as $1,234,567,89. The Edit instruction is provided for the purpose of formatting numeric output.



Edit Instruction

What is an Edit Instruction?

Edit is a very powerful instruction, in that it allows you greater control and flexibility in presenting numeric output. However, the power comes at a price. Edit is a very complex instruction. In order to use the full power of Edit, you must take the time to understand the details of how it works.

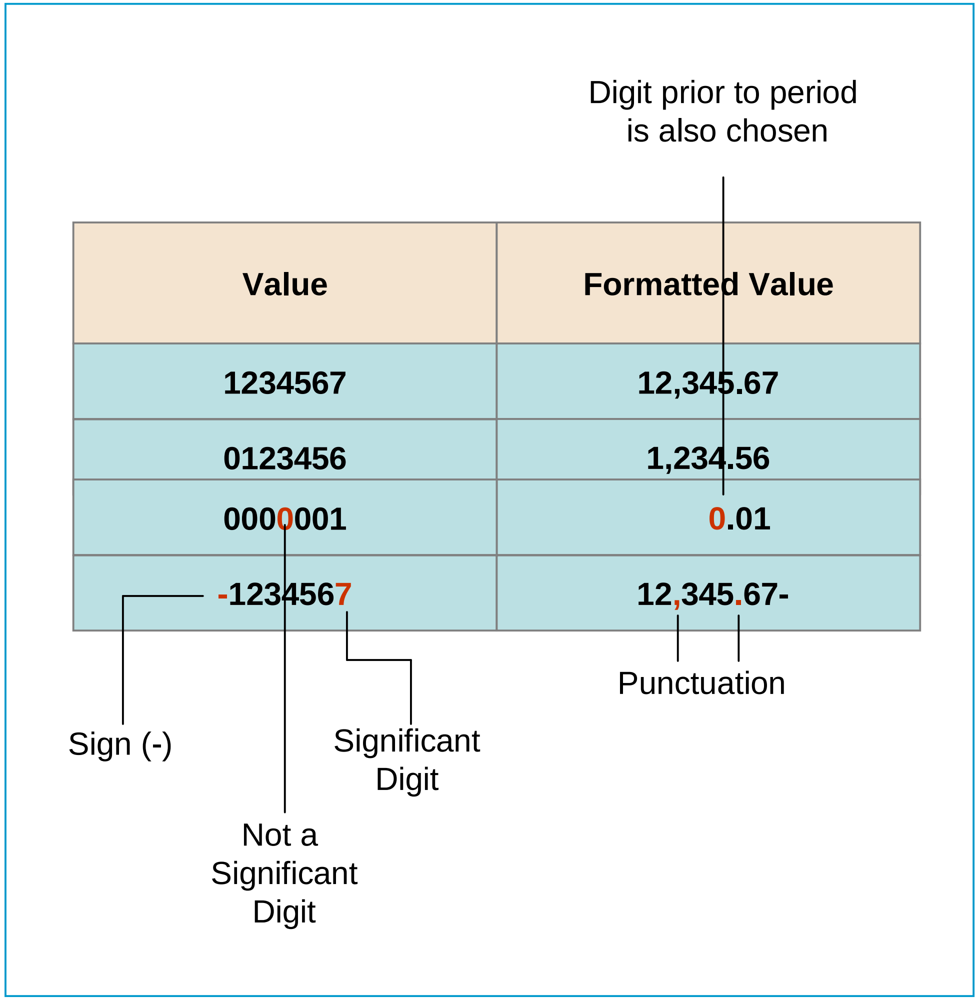
Value	Formatted Value
1234567	12,345.67
0123456	1,234.56
0012345	123.45
0001234	12.34
0000123	1.23
0000012	0.12
0000001	0.01
-1234567	12,345.67-

Edit

The values of all consist of seven digits and a sign. The positive sign is not shown.

The formatted values contain digits, and punctuations such as commas, periods and minus signs, to make reading the numbers easier. Note that the values are all 7 digits and a sign.

However, not all of the digits are significant in all of the examples. Zeroes preceding the first non-zero digit in a value are not significant.



Digit prior to period is also chosen

Value	Formatted Value
1234567	12,345.67
0123456	1,234.56
0000001	0.01
-1234567	12,345.67-

Punctuation

Sign (-)

Significant Digit

Not a Significant Digit

Continued…

Edit (cont’d)

If you were printing an amount on a check, you might want to format the amounts in dollars and cents to make it difficult to manually change the check.

One way to do this is to print asterisks rather than spaces prior to the first significant digit.

The table of formatted values would then look like the table shown on the right.

Value	Formatted Value
1234567	12,345.67
0123456	*1,234.56
0012345	***123.45
0001234	****12.34
0000123	*****1.23
0000012	*****0.12
0000001	*****0.01

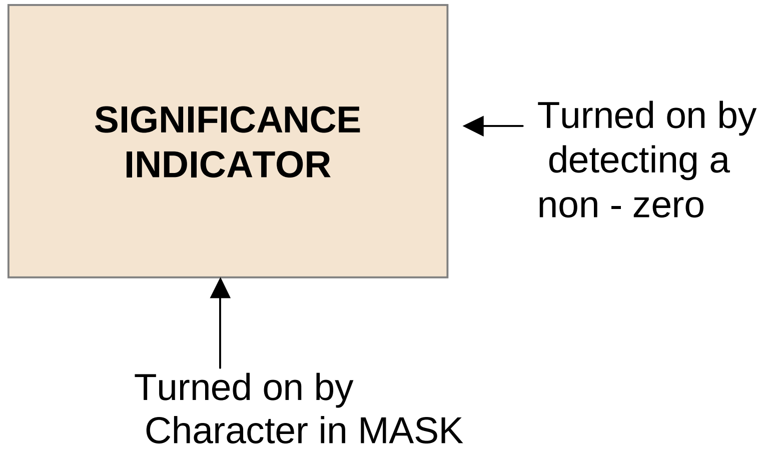
Significance Indicator

What is the Significance Indicator?

In order to handle this type of editing, Edit must be aware of whether the digit it is handling is significant or not. Edit maintains a Significance Indicator to keep track of whether a significant digit has yet been detected.

To be able to always print a digit before the decimal point, the programmer must be able to specify that Edit will behave as if it has found a significant digit once it reaches a certain position in the output field. This is known as forcing significance

Forcing Significance:

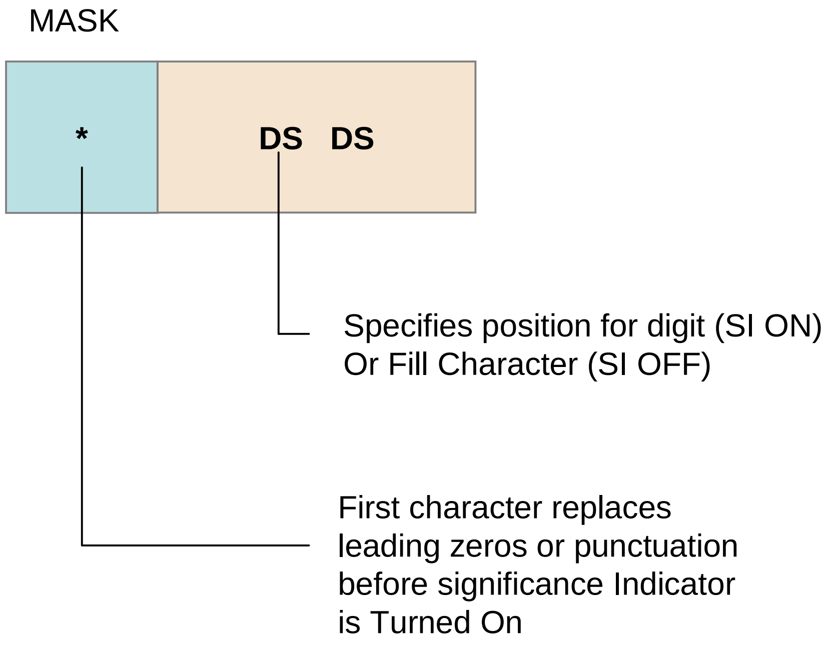

SIGNIFICANCE INDICATOR
Turned on by detecting a non - zero
Turned on by Character in MASK

Fill Character

What is the fill character?

The programmer must also be able to tell Edit what to place in the output field if the Significance Indicator has not been turned on. In the first table a blank is used, while in the second table, an asterisk is used. The character used to fill the output field, until the Significance Indicator is turned on, is known as the fill character.

Significance Indicator:



Specifies position for digit (SI ON)
Or Fill Character (SI OFF)

First character replaces
leading zeros or punctuation
before significance Indicator
is Turned On

Edit Mask

What is an edit mask?

The first operand of Edit is an edit mask that Edit replaces with the edited number. It is through the edit mask that the programmer specifies how the Edited number is to look. The first character of the edit mask is the fill character. The remaining characters of the edit mask are a combination of the characters shown in the table on the right.

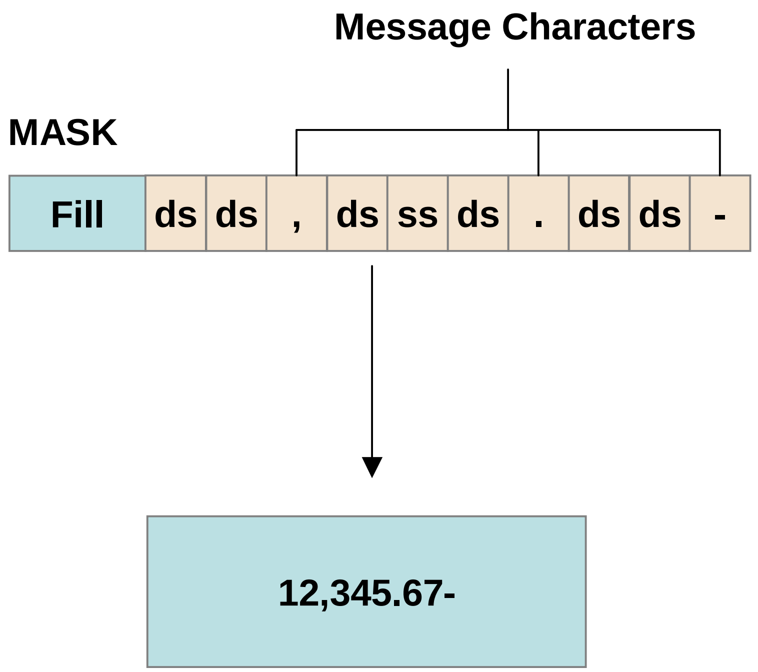
Character	Meaning
X ’20’	Digit Select
X ’21’	Significance Start
X ’22’	Field Separator
Anything else	Message Character

Continued…

Edit Mask (cont'd)

What are the components of an edit mask?

A digit select character specifies a position where a source digit is to be placed if the Significance Indicator is set, or the fill character is placed otherwise. A significance start character has the same effect as a digit select, but also forces the Significance Indicator on, after the digit or fill character is placed. The field separator character is only used if the Edit operation is editing multiple numbers. It causes the significance indicator to be turned off.

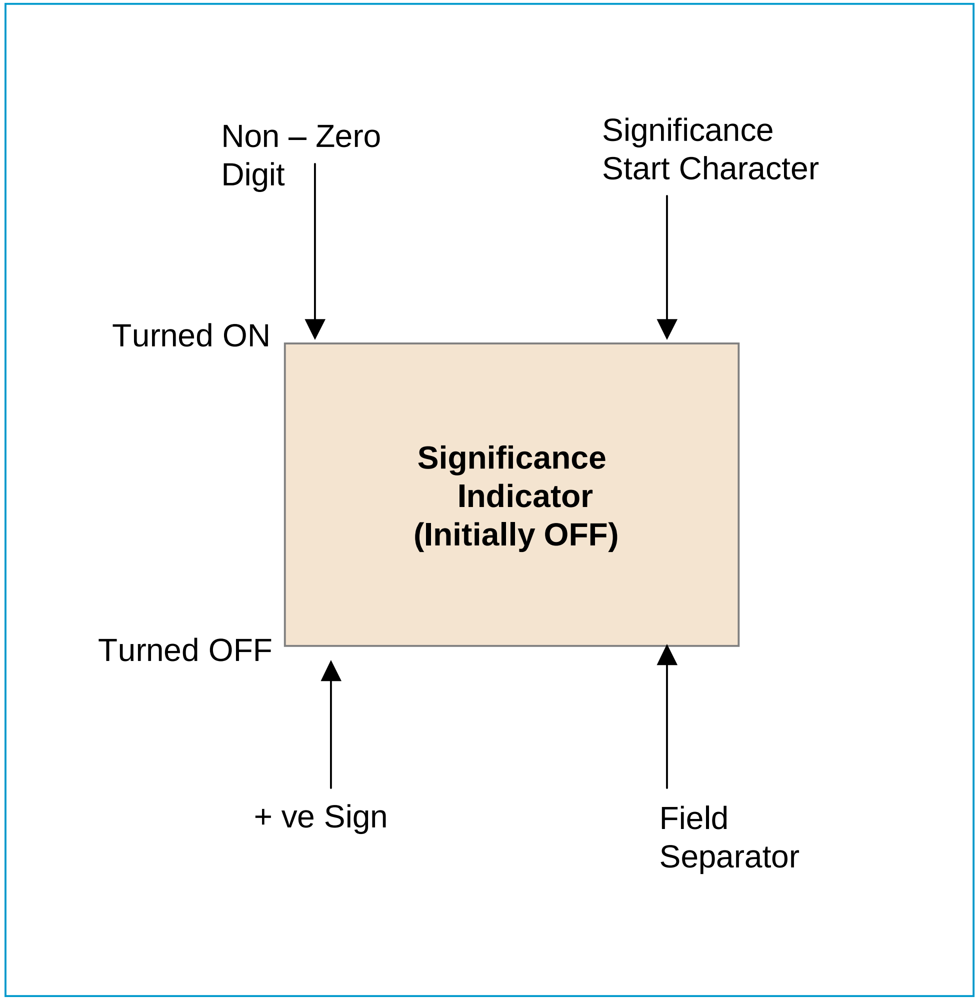


Continued…

Edit Mask (cont'd)

The Significance Indicator is initially turned off. It is turned on when a non-zero digit is encountered in the second operand, and after processing a byte in the mask containing a significance start character.

It is turned off when a positive sign is encountered in the second operand, and when a field separator character is found in the mask.



Formatting Packed Decimal Data with Edit Instruction

How do you format packed decimal data with Edit instruction?

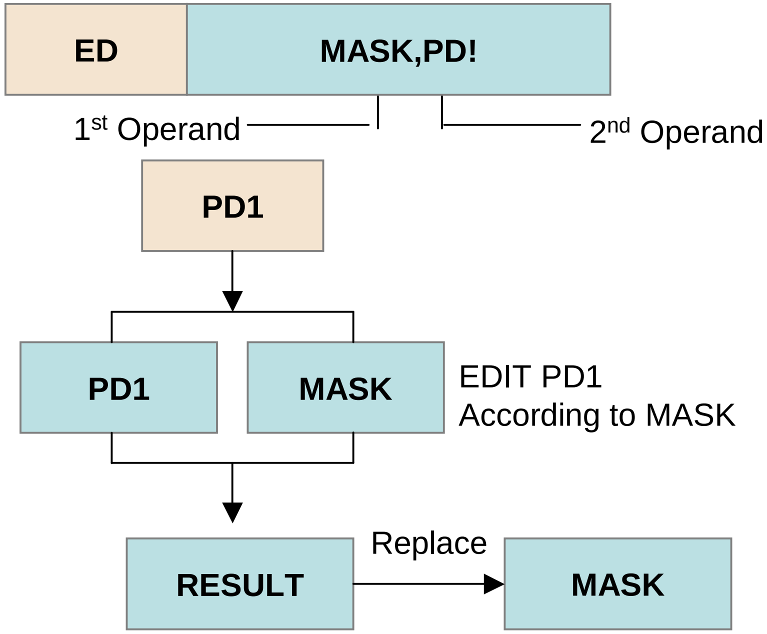
The Edit (ED) instruction is used to format packed decimal data. The first operand is an edit mask. The second operand is a field containing one or more packed decimal numbers.

The second operand is edited according to the values in the mask, and the edited number replaces the mask in the first operand field. The single length specifies the length of the mask. The two operands are processed left to right.

For example: ED MASK,PD1

The value in PDI is edited, according to the specifications in MASK, and the edited number replaces MASK.

Edit (ED)
SS Format (1 length) – Storage to Storage

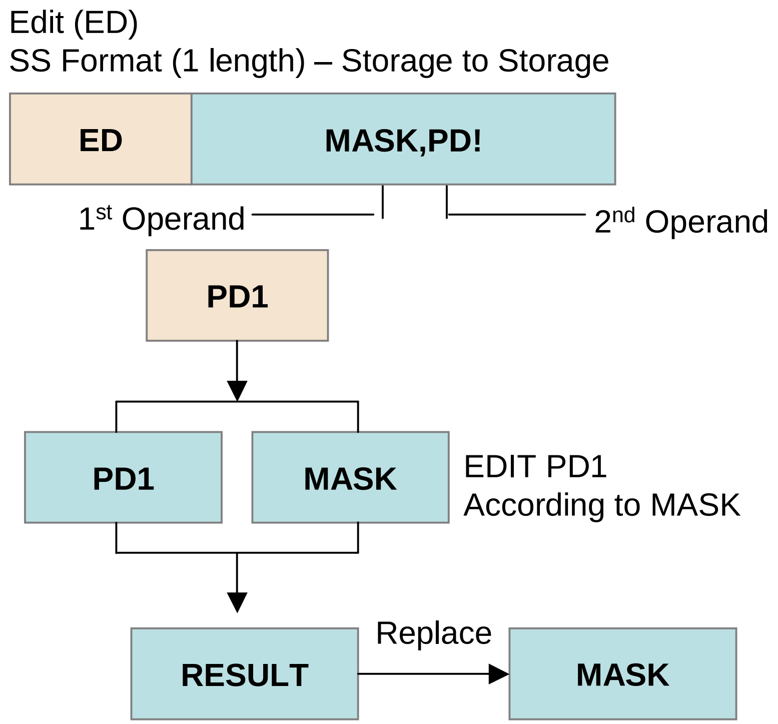


CC setting – 0 – the last field edited is zero or zero length.
1 – the last field edited is less than zero
2 – the last field edited is greater than zero

Continued…

Formatting Packed Decimal Data with Edit Instruction (cont'd)

The mask field is replaced by the edited number. It is necessary to move the mask to the location where the editing is to be done before each repeated ED instruction to avoid destroying the mask.



CC setting – 0 – the last field edited is zero or zero length.
1 – the last field edited is less than zero
2 – the last field edited is greater than zero

Continued…

Formatting Packed Decimal Data with Edit Instruction (cont'd)

To devise the mask necessary to edit the 7 digit number specified in the earlier example, you must remember that you will want the edited number to represent a dollar and cents amount.

If you look at this table, you can see that the fill character is blank. That blank will be the first character of the edit mask. If at least one digit is desired to be printed prior to the decimal point, you will need to specify a significance start character two positions before the decimal point, because the significance indicator is turned on after the byte with the significance start character is processed. From the last line of the table, it seems that the last character of the mask must be a minus sign.

Value	Formatted Value
1234567	12,345.67
0123456	1,234.56
0012345	123.45
0001234	12.34
0000123	1.23
0000012	0.12
0000001	0.01
-1234567	12,345.67-

Continued…

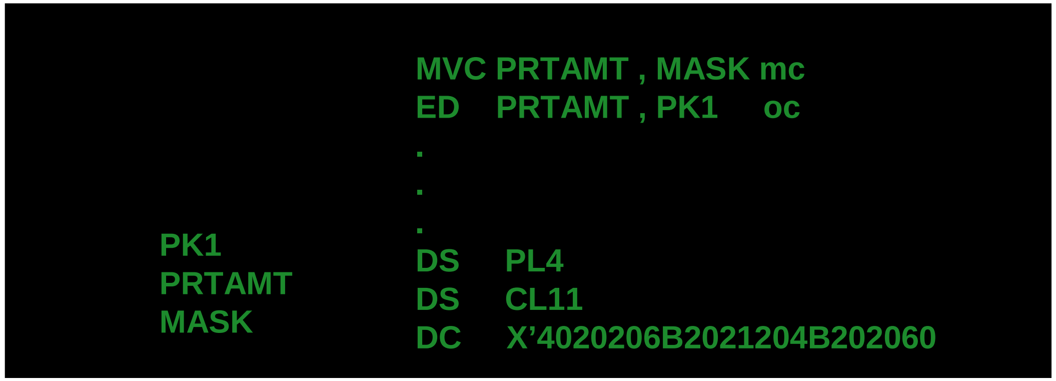
Formatting Packed Decimal Data with Edit Instruction (cont'd)

To edit a positive number, the positive sign turns the Significance Indicator off, so the fill character of blank would replace the minus sign message character. Using a minus sign does not turn the Significance Indicator off, so the message character is printed.

The mask will need to contain the following characters in the order shown in the table in the right.

Contents	Purpose
Blank	Fill character
X'20'	Digit select
X'20'	Digit select
Comma	Message character
X'20'	Digit select
X'21'	Significance start
X'20'	Digit select
Period	Message character
X'20'	Digit select
X'20'	Digit select
Minus sign	Message character

Digit Selects and Significance Starts

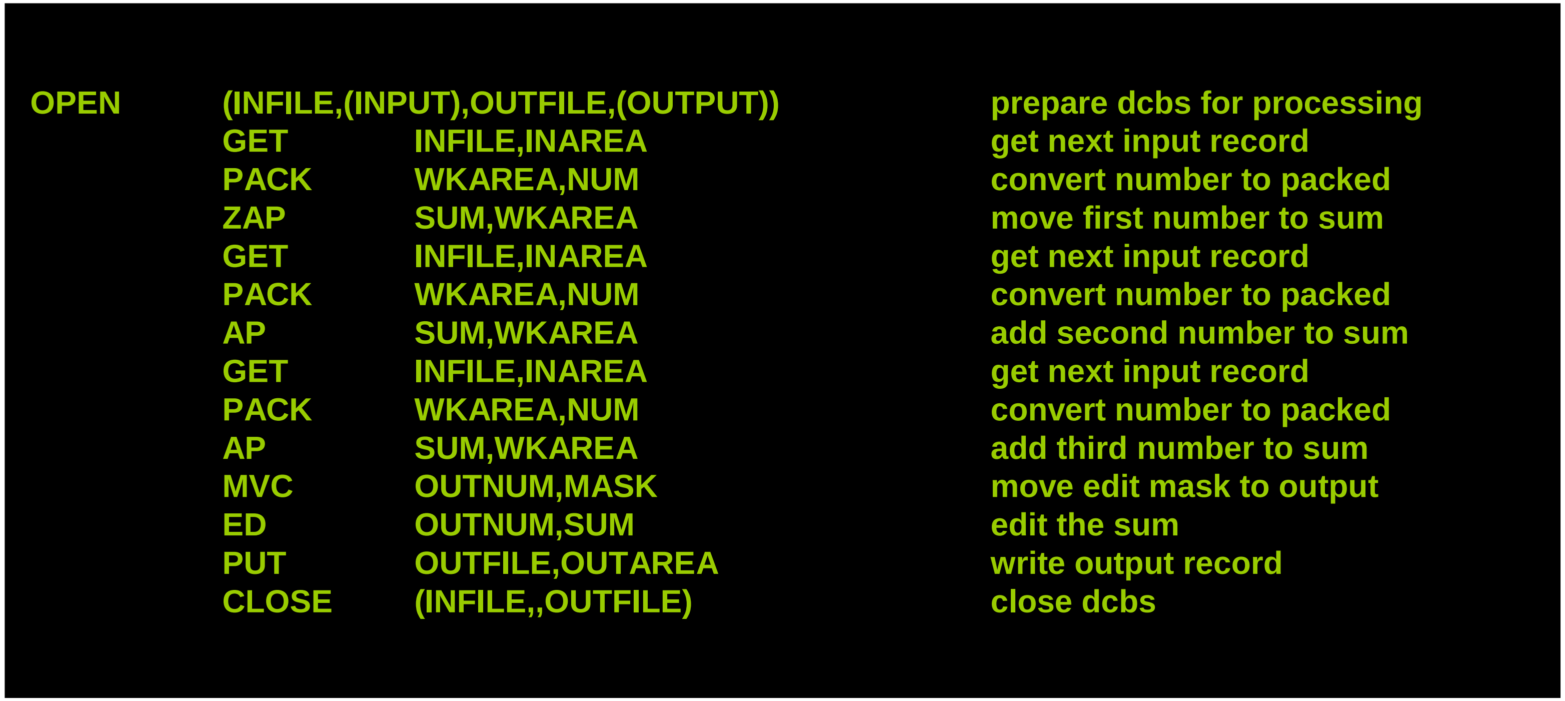


```
                 MVC PRTAMT , MASK mc
                 ED    PRTAMT , PK1     oc
                 .
                 .
                 .
PK1              DS     PL4
PRTAMT           DS     CL11
MASK             DC     X’4020206B2021204B202060
```

There are seven digits being edited in this example, so the total number of digit selects and significance start character is seven. When you add the fill character and message characters, the total length of the mask is 11 bytes. It is always a good idea to check that your masks have the proper number of digit selects and significance starts. Having too many may cause a data exception. Having too few will cause truncated values to be generated.

Assuming that the number to be edited was called PK1 and the field where you wanted to place the edited number was called PRTAMT, the code to do the editing is as shown above.

EBCDIC Code

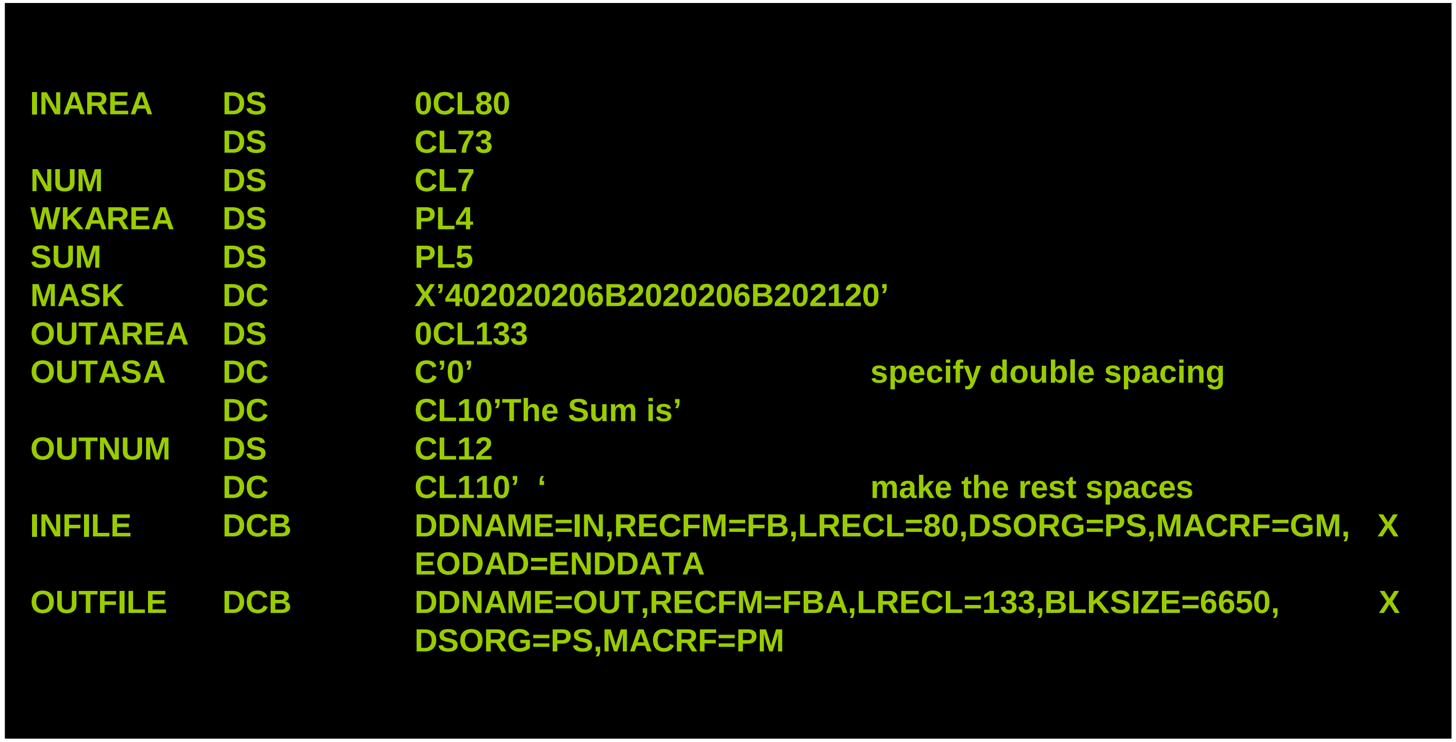


```
OPEN     (INFILE,(INPUT),OUTFILE,(OUTPUT))         prepare dcbs for processing
         GET      INFILE,INAREA                    get next input record
         PACK     WKAREA,NUM                       convert number to packed
         ZAP      SUM,WKAREA                       move first number to sum
         GET      INFILE,INAREA                    get next input record
         PACK     WKAREA,NUM                       convert number to packed
         AP       SUM,WKAREA                       add second number to sum
         GET      INFILE,INAREA                    get next input record
         PACK     WKAREA,NUM                       convert number to packed
         AP       SUM,WKAREA                       add third number to sum
         MVC      OUTNUM,MASK                      move edit mask to output
         ED       OUTNUM,SUM                       edit the sum
         PUT      OUTFILE,OUTAREA                  write output record
         CLOSE    (INFILE,,OUTFILE)                close dcbs
```

The EBCDIC codes for blank, comma, period and minus are 40, 6B, 4B and 60 respectively. These are codes that you should memorize if you are going to be doing a lot of editing.

Continued…

EBCDIC Code (cont’d)



```
INAREA     DS       0CL80
           DS       CL73
NUM        DS       CL7
WKAREA     DS       PL4
SUM        DS       PL5
MASK       DC       X’402020206B2020206B202120’
OUTAREA    DS       0CL133
OUTASA     DC       C’0’                        specify double spacing
           DC       CL10’The Sum is’
OUTNUM     DS       CL12
           DC       CL110’ ‘                    make the rest spaces
INFILE     DCB      DDNAME=IN,RECFM=FB,LRECL=80,DSORG=PS,MACRF=GM,   X
                    EODAD=ENDDATA
OUTFILE    DCB      DDNAME=OUT,RECFM=FBA,LRECL=133,BLKSIZE=6650,     X
                    DSORG=PS,MACRF=PM
```

A complete code segment, to read 3 seven digit numbers from the first seven positions of 3 in a dataset, add them and then print the edited sum, is shown above.

Load Operation

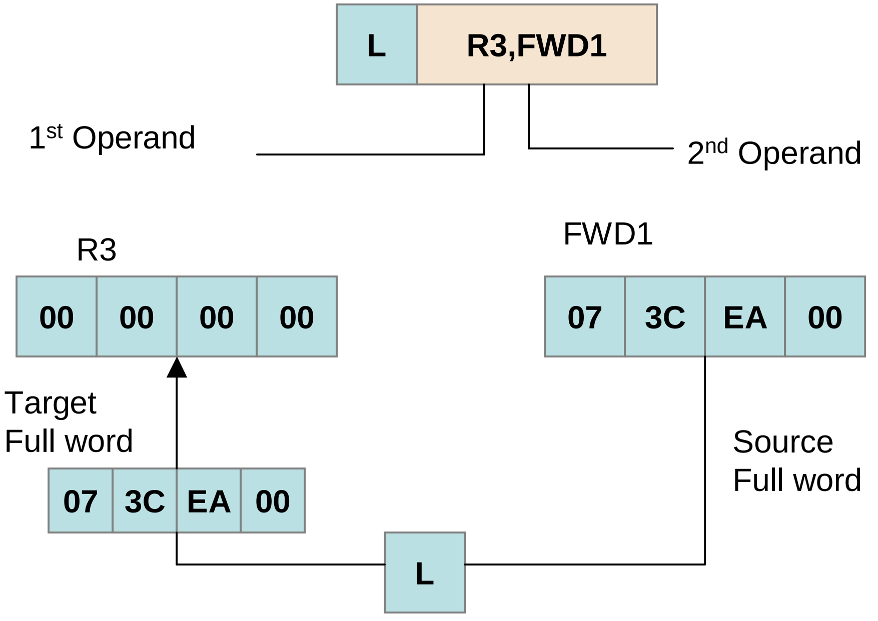
A load operation moves data from main storage into a GPR. You will find this useful when performing fixed-point binary arithmetic, or using data for addressing. The first two instructions you will learn are Load (L) and Load Halfword (LH).

The Load (L) instruction is used to move a fullword of data from main storage into a General Purpose Register (GPR). The data is moved unchanged, from storage to the register.

For example: L R3,FWD1

The fullword of data at location FWD1 would be loaded into R3. The data at location FWD1 is unchanged.

Load (L) Instruction:
RX Format – Register and Indexed Storage



Copies FULWORD from main Storage address beginning at location FWD1 to GPR R3

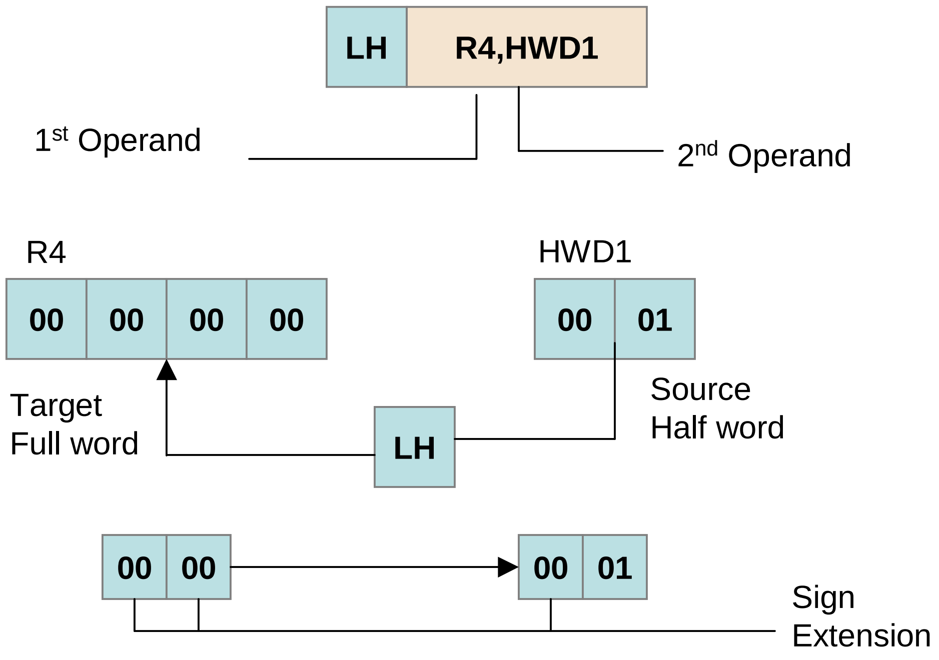
CC setting: Condition Code is unchanged

Load Halfword Instruction

The Load Halfword (LH) instruction takes a halfword of data from main storage HWD1, expands it to a fullword by sign extension, and places the resultant fullword into a GPR R4.

For example: LH R4,HWD1

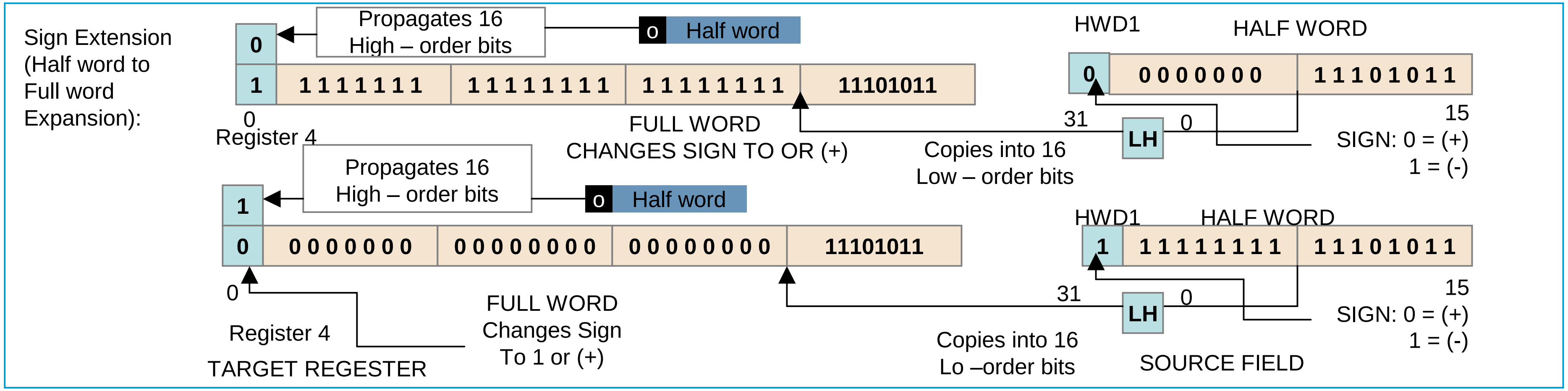
Load Half word (LH):
RX Format – Register and Indexed Storage



Copies HALFWORD from main storage address beginning at location HWD! To GPR r4 (low – order 16 bits). Since The sign of HWD1 is 0, HWD1 is expanded into a full word By inserting 16 o bits in front of the 16 bits of HWD1
CC setting: Condition Code is unchanged

Note! The expansion occurs within the processor and does not alter any values in the main storage.

Sign Extension



Sign Extension (Half word to Full word Expansion):

The example above shows how sign extension works. The high order bit of a fixed-point number is its sign, and the sign bit of the source field directly affects the high-order 16 bits in the target register.

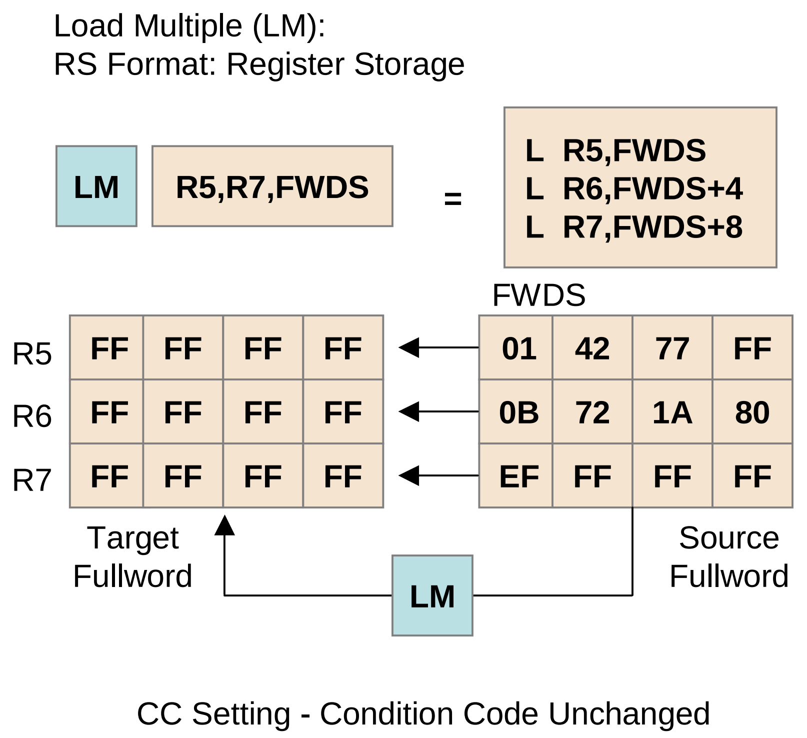
Load Multiple Instruction

What is the Load Multiple (LM) Instruction?

This operation could be performed by a number of load instructions, each loading a single fullword into a register one at a time. However the Load Multiple (LM) instruction is provided to facilitate this fairly common requirement by a single instruction.

Using the LM instruction, a number of consecutive fullwords from main storage are loaded into a range of GPRs, from the GPR specified as the first operand to the GPR specified as the second Operand, both inclusive.

For example: LM R5,R7,FWDS



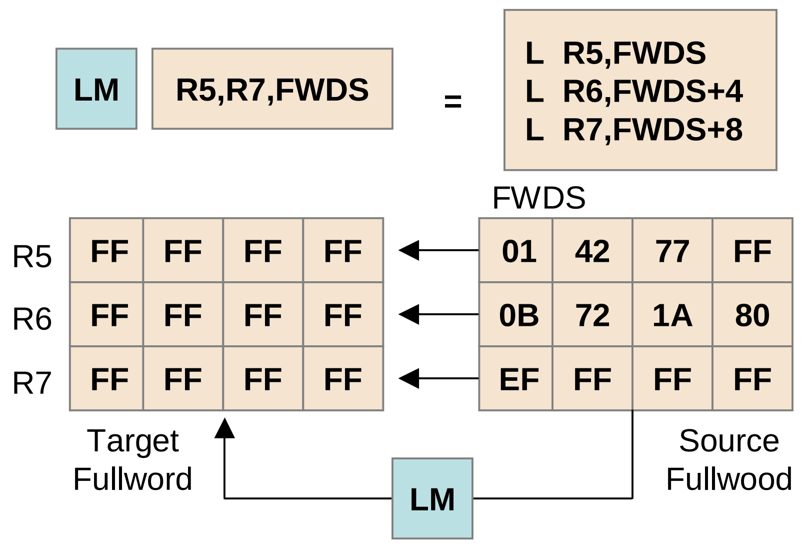
Continued…

Load Multiple Instruction (cont'd)

When the instruction executes, it loads a fullword from main storage at location FWDS to the GPRs beginning with R5, the next fullword in storage (i.e. at location FWDS+4) would be loaded into R6 and the next (at location FWDS+8) into R7.

The effect of LM R5,R7,FWDS is identical to the three instructions shown on the right.

Load Multiple (LM):
RS Format: Register Storage



R5	FF	FF	FF	FF
R6	FF	FF	FF	FF
R7	FF	FF	FF	FF

FWDS			
01	42	77	FF
0B	72	1A	80
EF	FF	FF	FF

CC Setting - Condition Code Unchanged

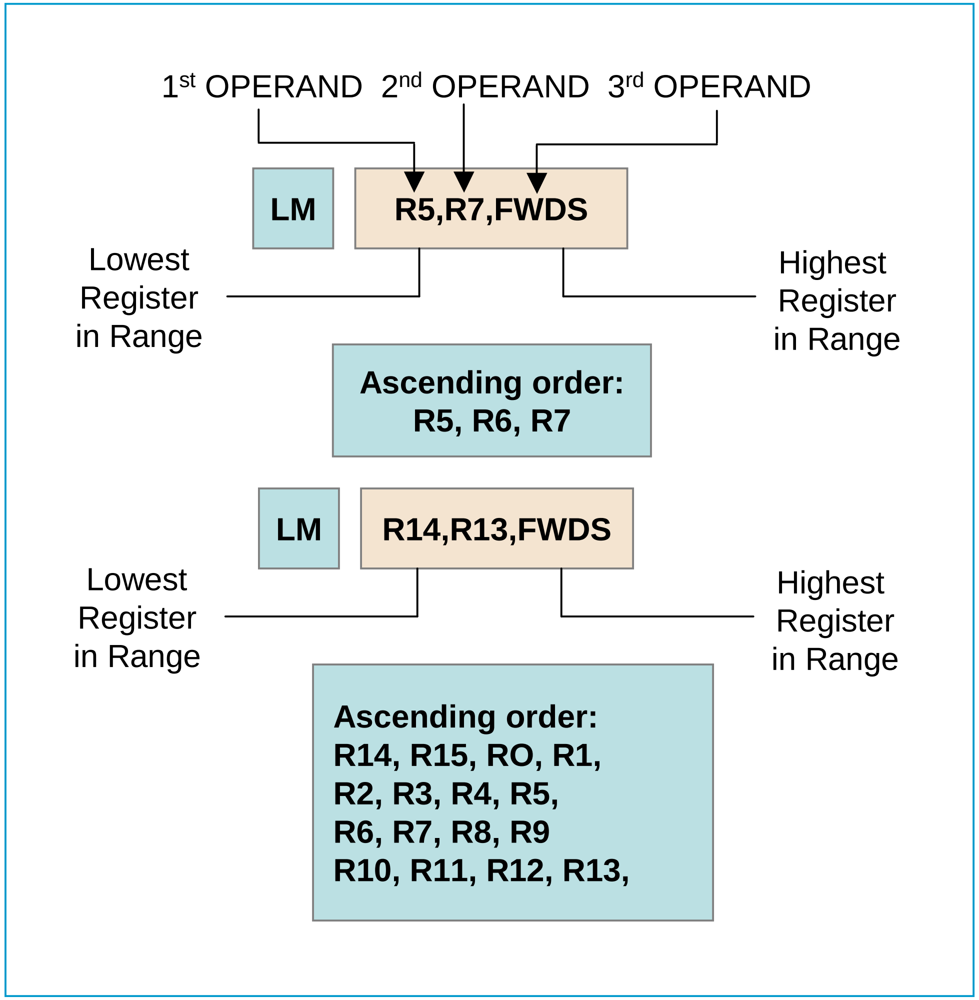
Continued…

Load Multiple Instruction (cont'd)

Load Multiple (LM) is a Register Storage format instruction.

LM has three operands:

- The first operand is the first in a group of consecutive target registers
- The second operand specifies the last register in the group
- The third operand specifies the source main storage address


1st OPERAND
2nd OPERAND
3rd OPERAND
LM
R5,R7,FWDS
Lowest Register in Range
Highest Register in Range
Ascending order: R5, R6, R7
LM
R14,R13,FWDS
Lowest Register in Range
Highest Register in Range
Ascending order: R14, R15, RO, R1, R2, R3, R4, R5, R6, R7, R8, R9 R10, R11, R12, R13,

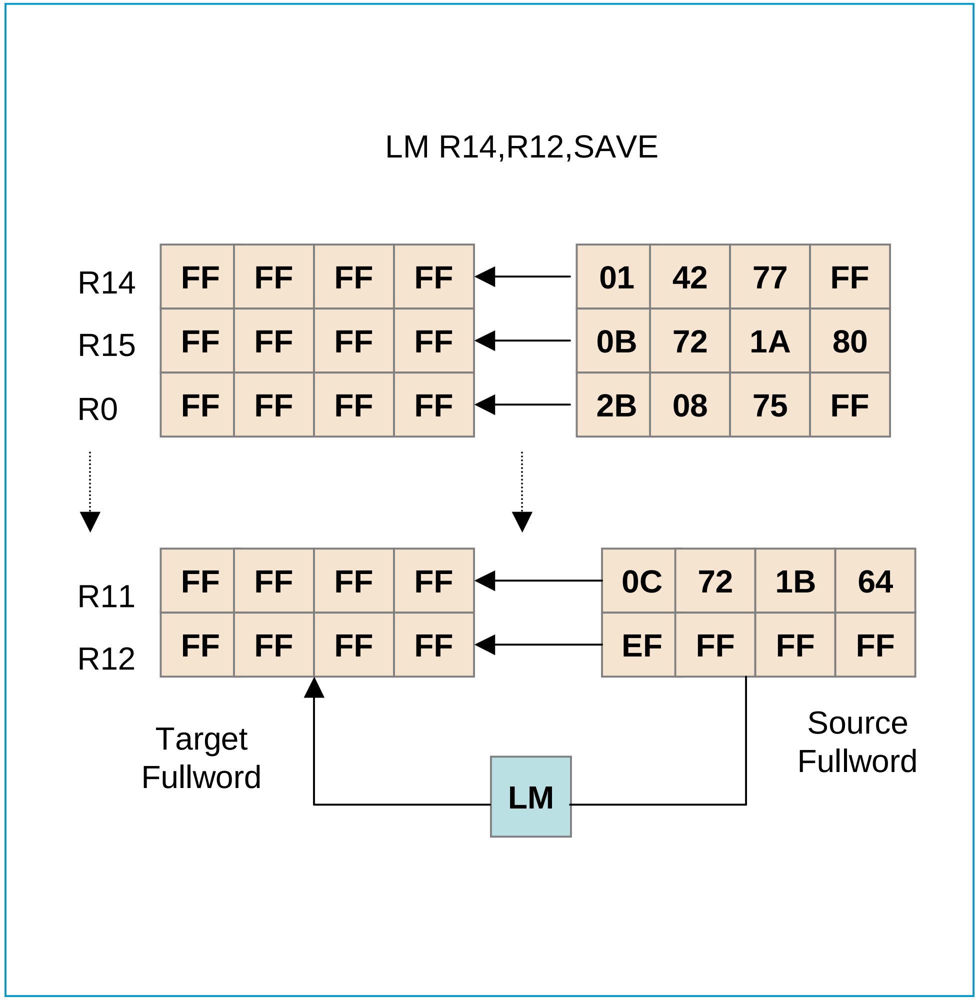
Continued…

Load Multiple Instruction (cont’d)

One interesting feature of Load Multiple (LM) is that if the second operand specifies a lower numbered register than the first operand, this indicates a range of registers from R14 to R15 and from R0 to R12.

Thus LM R14,R12,SAVE

Loads 15 registers; R14, R15, R0, R1….R12 in that order, from the 15 consecutive fullwords beginning at SAVE.



LM R14,R12,SAVE

R14	FF	FF	FF	FF	01	42	77	FF
R15	FF	FF	FF	FF	0B	72	1A	80
R0	FF	FF	FF	FF	2B	08	75	FF
R11	FF	FF	FF	FF	0C	72	1B	64
R12	FF	FF	FF	FF	EF	FF	FF	FF

Target Fullword

LM

Source Fullword

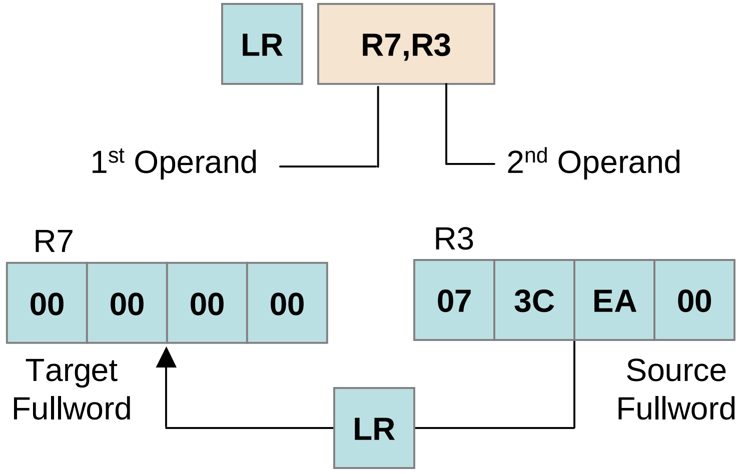
Load Register

The two instructions LR and LTR covered in this section are both RR (register to register) type. That is, both of their operands specify registers. The first operand specifies the target register, and the second operand specifies the source register.

The Load Register (LR) instruction copies a fullword from one GPR (2nd Operand) to another GPR (1st Operand).

For example: LR R7,R3

Load Multiple (LR):
RR Format - Register to Register



CC Setting: Condition Code is Unchanged

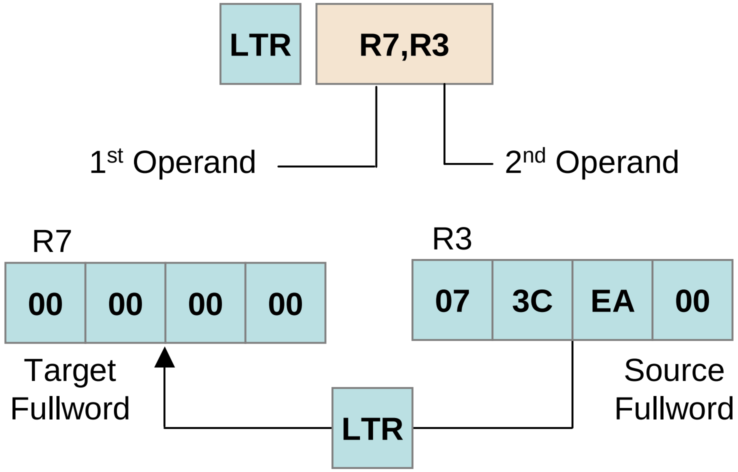
Load and Test Instruction

The Load and Test (LTR) instruction is identical to Load Register (LR), except that it sets the condition code.

The instruction copies the contents of one GPR (2nd Operand) to another GPR (1st Operand), setting the CC to indicate if the result is zero, negative or positive.

For example: LTR R7,R3

Load and Test (LTR):
RR Format - Register to Register



LTR R7,R3

1st Operand

2nd Operand

R7

00 00 00 00

Target
Fullword

R3

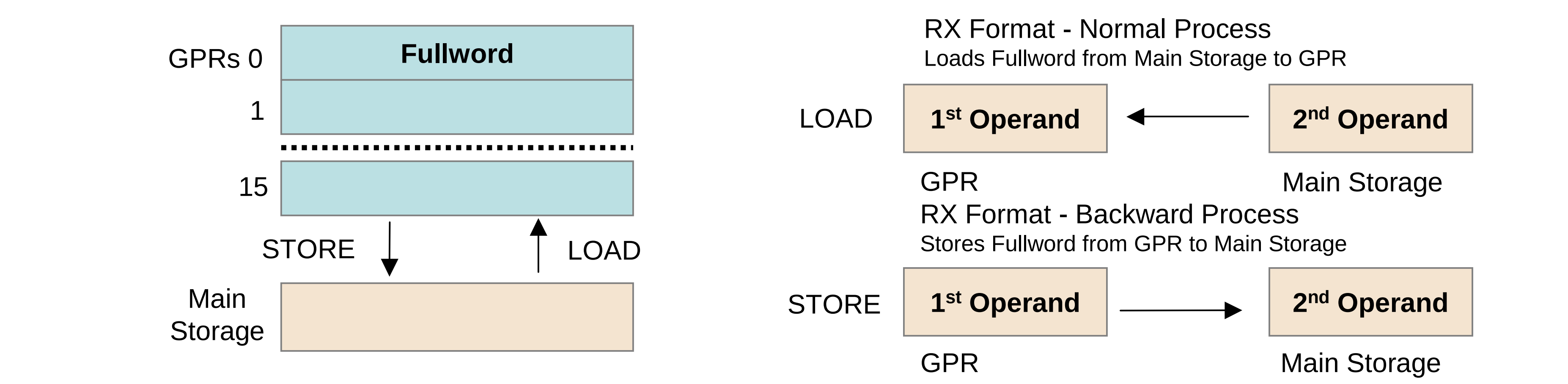
07 3C EA 00

Source
Fullword

LTR

CC Setting – 0-Result is Zero
1-Result is Negative
2-Result is Positive

Store and Store Halfword Operations


GPRs 0
Fullword
1
15
STORE
LOAD
Main Storage
RX Format - Normal Process
Loads Fullword from Main Storage to GPR
LOAD
1st Operand
2nd Operand
GPR
Main Storage
RX Format - Backward Process
Stores Fullword from GPR to Main Storage
STORE
1st Operand
2nd Operand
GPR
Main Storage

This section describes two operations: Store (ST) and Store Halfword (STH). A store operation is the opposite of a Load (L); it is a move of data from a register to main storage. The store instructions are the first examples of instructions you will see that work backwards. That is, the result of a store operation is placed in the second operand field, rather than the first as is normal.

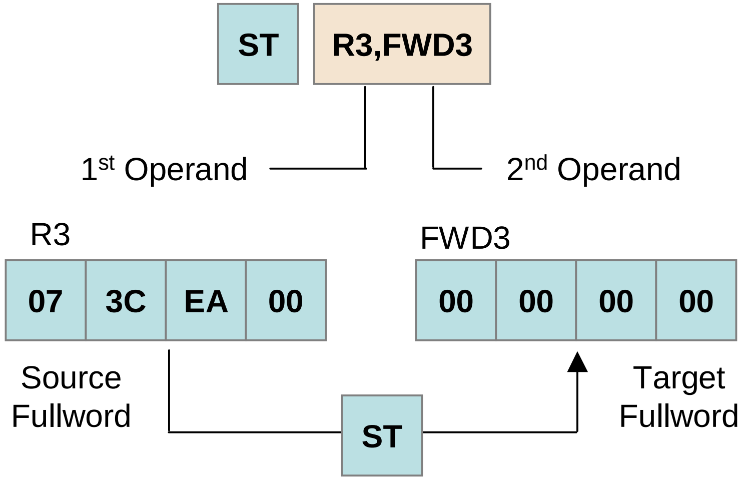
Store Instruction

The Store (ST) instruction is similar to the Load (L). ST stores a fullword, located in a GPR, unchanged into a specified main storage location.

For example: ST R3,FWD3

The ST instruction stores a fullword from the GPR (1st Operand) R3 into a fullword in indexed storage (2nd Operand) at location FWD3.

Store (ST):
RX Format - Register and Indexed Storage



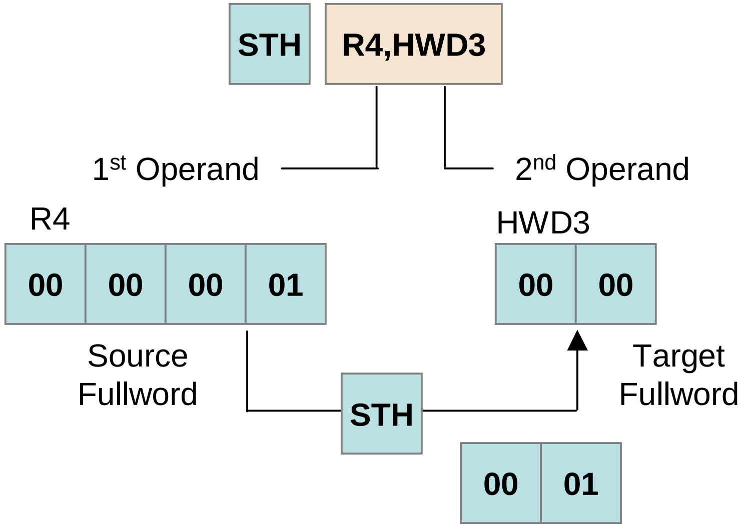
CC setting: Condition Code is unchanged

Store Halfword Instruction

The Store Halfword (STH) instruction stores the Low- order 16 bits (halfword) of a GPR into a Halfword location in main storage. The high order 16 bits are truncated, with no indication if significant digits are truncated or the sign is changed.

For example: STH R4,HWD3

Store Halfword (STH):
RX Format - Register and Indexed Storage



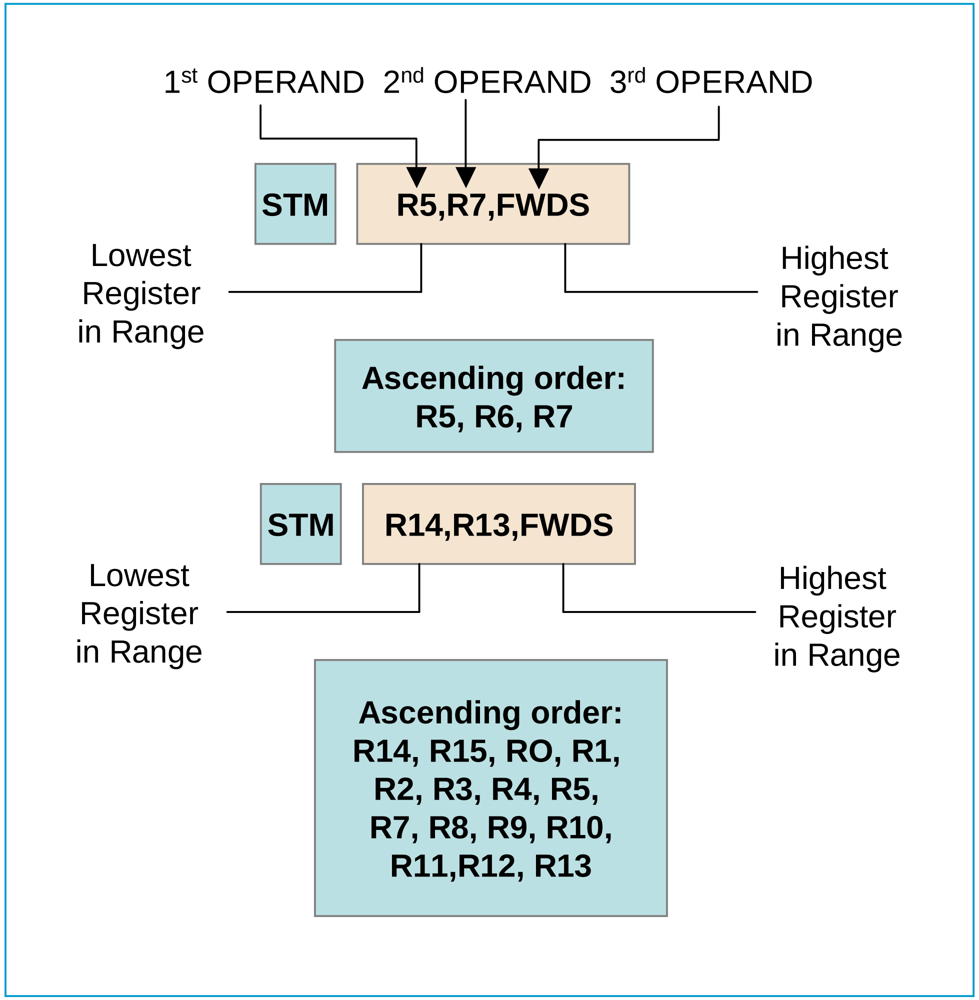
CC setting - Condition Code is unchanged

Store Multiple Register

Store Multiple (STM) is a RS format (register to Storage) instruction. The STM instruction performs the opposite operation of the LM instruction.

STM has three operands:

- The first operand specifies the starting register number
- The second operand specifies the last register in the group
- The third operand specifies the target main storage address


1st OPERAND
2nd OPERAND
3rd OPERAND
STM
R5,R7,FWDS
Lowest Register in Range
Highest Register in Range
Ascending order: R5, R6, R7
STM
R14,R13,FWDS
Lowest Register in Range
Highest Register in Range
Ascending order: R14, R15, RO, R1, R2, R3, R4, R5, R7, R8, R9, R10, R11,R12, R13

Continued…

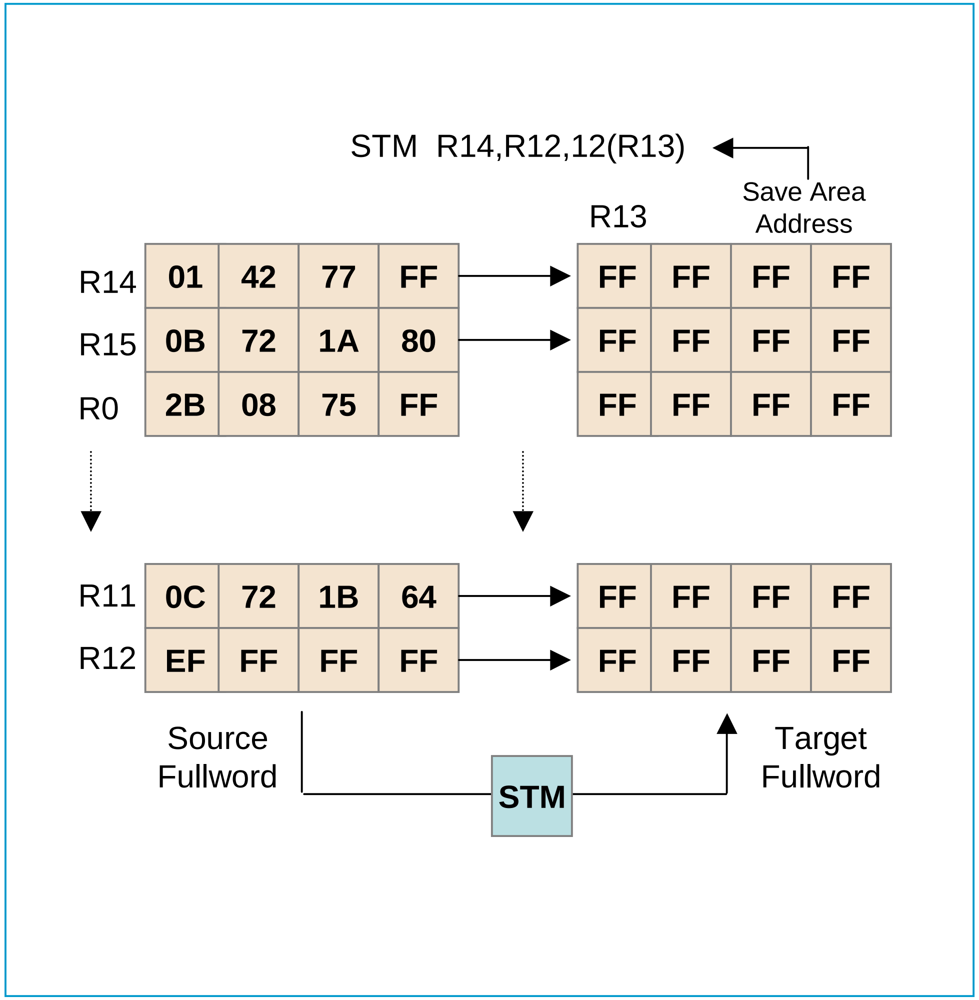
Elements of the Topic

What is the function of the Store Multiple Instruction?

The Store Multiple (STM) instruction is used as part of the standard housekeeping done at the beginning of a program. This is performed to save the register contents of a program's caller, so they can later be restored before returning control to the caller.

In fact, the first instruction frequently used is a STM to perform this operation. An example is shown on the right.

For example: STM R14,R12,12(R13)



Insert Character Instruction

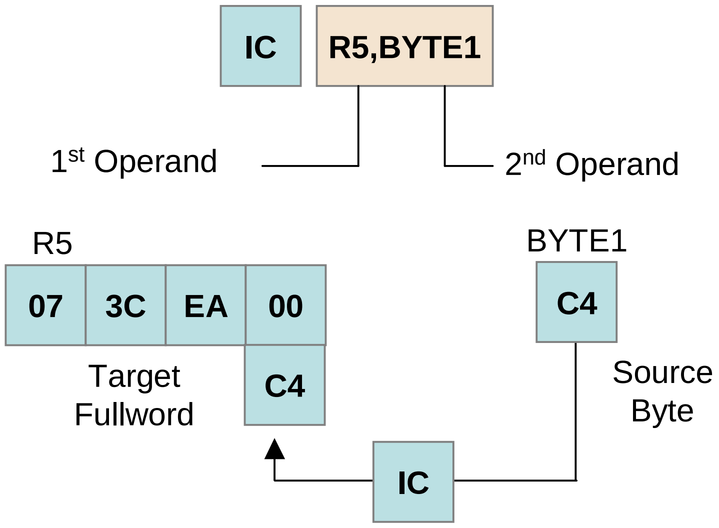
The load and store instructions presented in this course all work with either fullwords or halfwords. There are also instructions that transfer bytes of data between main storage and GPRs.

What is the function of the IC Instruction?

The Insert Character (IC) instruction transfers a single byte of data from main storage to the low order 8 bit positions of the register specified. The high order 24 bits of the register are unchanged.

For example: IC R5,BYTE1

Insert Character (IC):
RX Format Register Index Storage



CC Setting - Condition Code is Unchanged

Store Character Instruction

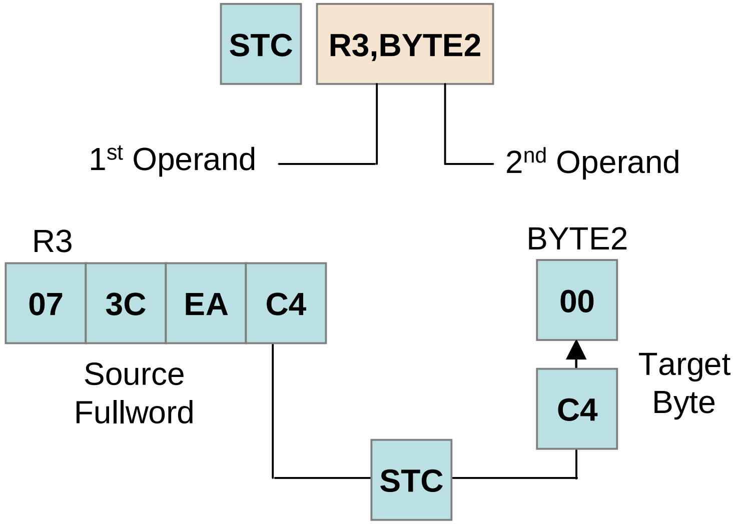
What is the function of the Store Character (STC) Instruction?

The Store Character (STC) instruction stores the low order byte of a GPR to a byte in main storage. Note that it is the second operand that receives the result.

For example: STC R3,BYTE2

The low order byte of GPR (1st Operand) R3 is placed in Indexed storage (2nd Operand) at location BYTE2.

Store Character (STC):
RX Format - Register to Index Storage



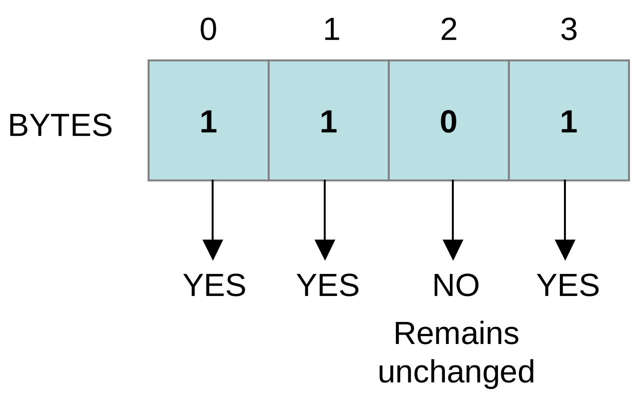
CC Setting - Condition Code is Unchanged

ICM and STCM Instructions

The IC and STC instructions are limited in that they process only a single byte of data, and only use the low order byte of the GPR specified as the first operand. The Insert Characters under Mask (ICM) and Store Characters under Mask (STCM) instructions provide more flexibility.

These two instructions, ICM and STCM, are RS format register storage instructions. The second operand for these instructions is a mask field, not a GPR. The 4 bits of the mask field correspond, one for one, with the four bytes of the GPR specified as the first operand.

$13_{10} = 1101_2$ is translated to
The mask field below



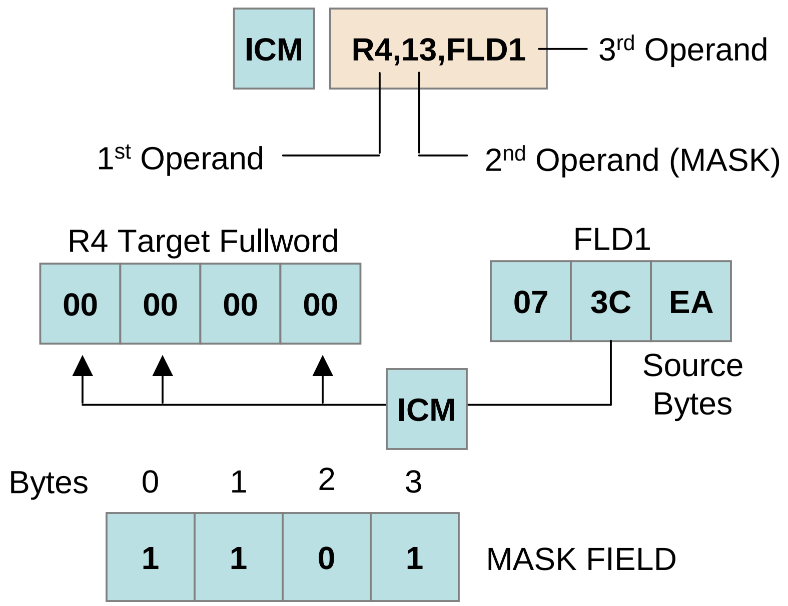
Insert Characters Under Mask Instruction

What is the function of the ICM Instruction?

The Insert Characters Under Mask (ICM) instruction is used for logical bit and byte operations. It inserts consecutive bytes starting at the third operand address, into positions of the GPR specified as the first operand for which the corresponding mask bits are one. Other bytes in the first operand are unchanged.

For example: ICM R4,13,FLD1

Insert Characters Under Mask (ICM):
RS Format - Register Storage



CC Setting - 0 Mask is zero or all inserted bits are zero
1 Leftmost inserted bit is one
2 Leftmost inserted bit is zero and not all inserted bit zeros

Store Characters Under Mask Instruction

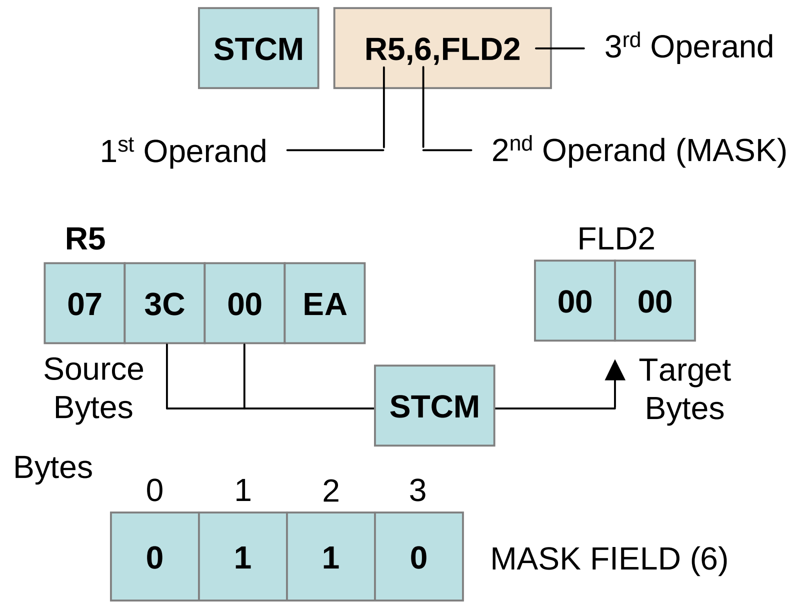
What is the function of the Store Characters Under Mask Instruction?

The Store Characters Under Mask (STCM) instruction is used for logical bit and byte operations.

This instruction stores the contents of those byte positions of the register indicated by a one bit in the mask, to consecutive main storage locations starting at the address specified as the third operand.

For example : STCM R5,6,FLD2

Store Characters Under Mask (STCM):
RS - Register Storage



CC Setting - Condition Code is Unchanged

Move Characters Instruction

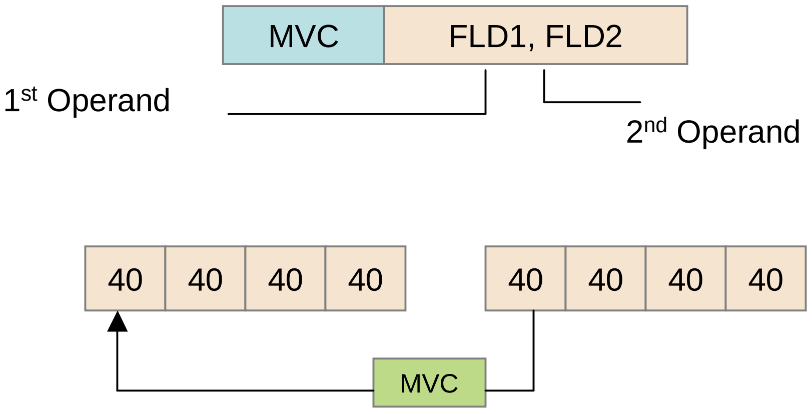
Move Characters (MVC): SS(1 length) Storage to Storage. The actual number of bytes to be copied is Indicated by the first operand.

MVC	FLD1, FLD2

When the format is used as shown above the Assembler supplies the understood length code.

MVC	FLD1(4), FLD2

The Length can be explicitly stated as shown above. Then length is 4 byes. CC Setting – Condition Code is Unchanged.



The Move Characters (MVC) instruction is used to move data from one location in main storage to another. It is a one-length SS instruction, so up to 256 bytes of data can be moved with a single MVC instruction. Data is moved, one byte at a time, left to right from the second operand location to the first.

For example: MVC FLD1,FLD2

Move Immediate Instruction

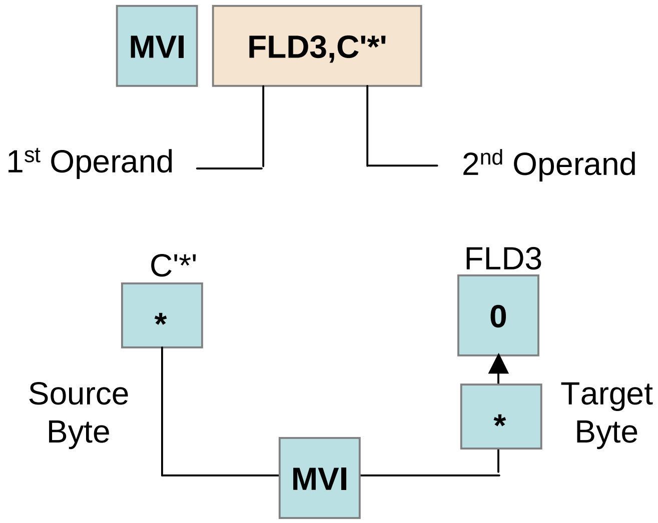
What is the function of the Move Immediate (MVI) Instruction?

The Move Immediate (MVI) instruction is in SI format. It moves its immediate operand, the one byte operand contained within the instruction itself, to the first operand location.

For example: MVI   FLD3,C'*'

This instruction moves an asterisk to FLD3, a one-byte field.

Move Immediate (MVI):
SI Format - Storage Immediate Data

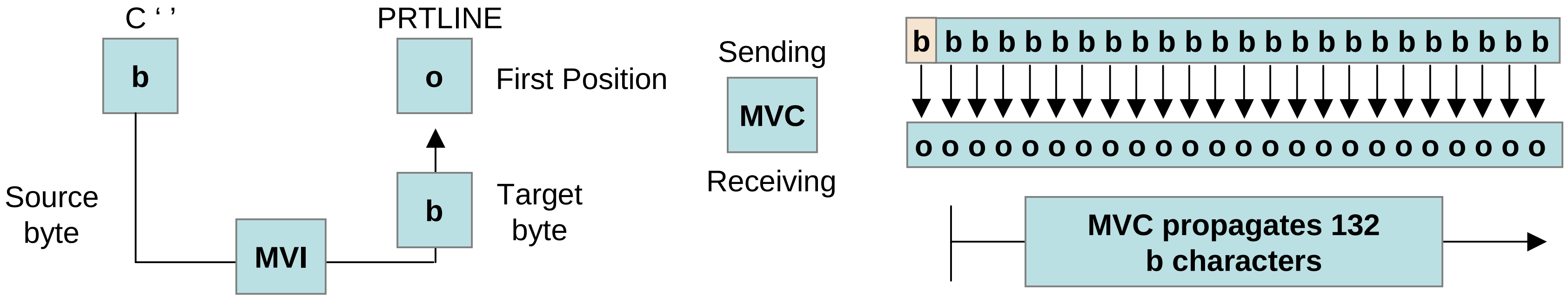


Copies the Immediate Data '*' (one byte second operand field) to the first operand (one bite field)

CC Settings - Condition Code is Unchanged

Move Characters Instruction

```
MVI  PRTLINE,C ' '                 MOVE BLANK TO FIRST POSTION
MVI  PRTLINE+1(132),PRTLINE        PROPAGATE BLANK THROUGH FIELD
```

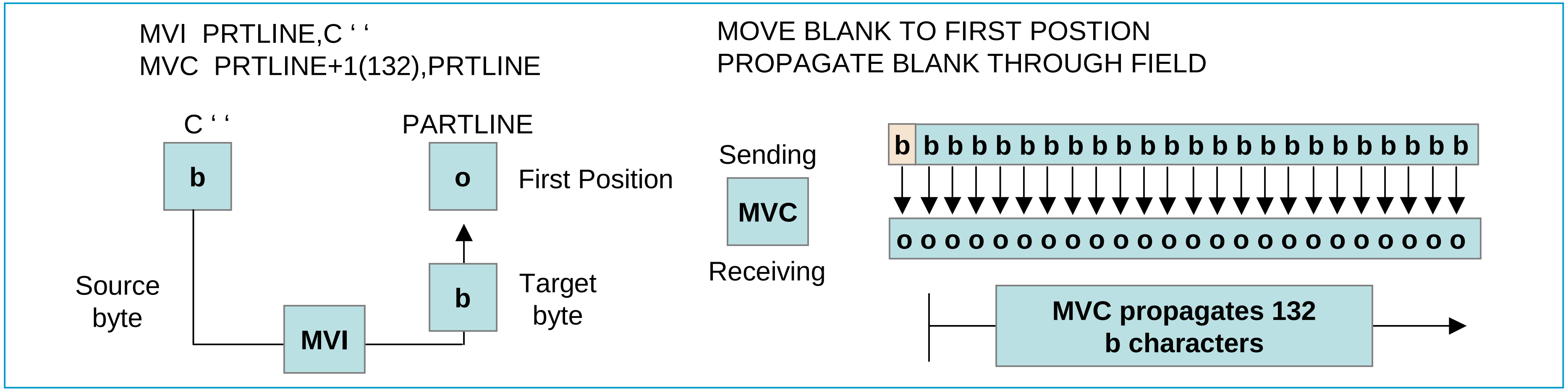

C ' '
b
Source byte
MVI
PRTLINE
o
First Position
b
Target byte
Sending
MVC
Receiving
b b b b b b b b b b b b b b b b b b b b b b b b
o o o o o o o o o o o o o o o o o o o o o o o o
MVC propagates 132 b characters

You can take advantage of knowing that MVC works left to right, one byte at a time, to produce an initialization technique that does not require a long constant. Consider the code above.

The MVI instruction sets the first byte of PRTLINE to blank. Consider the operation of the MVC Instruction byte by byte. It works byte by byte, left to right, so the first byte that gets moved is the byte at PRTLINE to the destination, PRTLINE+1. Now, both, PRTLINE and PRTLINE+1 contain blanks.

Continued…

Move Characters Instruction (cont’d)


MVI PRTLINE,C ‘ ‘
MVC PRTLINE+1(132),PRTLINE
MOVE BLANK TO FIRST POSTION
PROPAGATE BLANK THROUGH FIELD
C ‘ ‘
b
PARTLINE
o
First Position
Source byte
MVI
b
Target byte
Sending
MVC
Receiving
b b b b b b b b b b b b b b b b b b b b b b b b
o o o o o o o o o o o o o o o o o o o o o o o o
MVC propagates 132 b characters

The next byte is moved from PRTLINE+1 to PRTLINE+2. Now, PRTLINE+2 contains a blank. The destination field on each cycle becomes the sending field on the next. The MVI moves in the first blank to the leftmost position.

The MVC moves the other 132 bytes, one by one, down through the field. The whole field is initialized, without defining a long field to contain a constant.

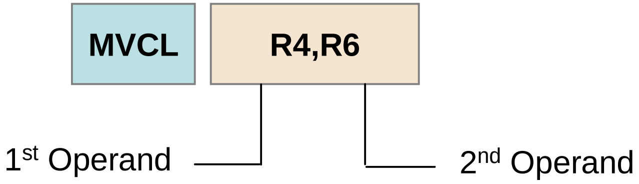
Move Long Instruction

The MVC instruction is only capable of moving up to 256 bytes. There are many situations when a programmer wants to move much larger blocks of data. It could be done with multiple MVCs, but that would be tedious, and not very efficient.

What is the function of the Move Long Instruction?

The Move Long (MVCL) instruction allows moving up to 16,777,215 bytes of data with a single instruction. The first thing you will notice is that although MVCL is an instruction that specifies movement of one storage operand to another, it is an RR format instruction.

Move Long (MVCL):
RR Format - Register to Register



CC Setting - 0 - Operand lengths equal.
1 - First operand length low
2 - Second Operand length high.
3 - Destructive Overlap, No Movement.

Continued…

Move Long Instruction (cont'd)

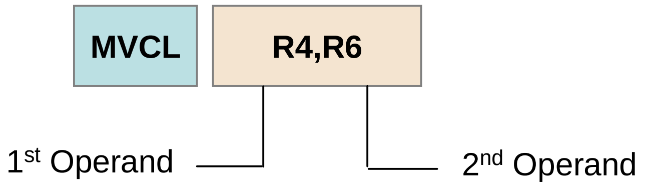
The MVCL Instruction moves the content of the memory location whose address is in GPR (2nd operand) R6 to the memory location whose address is in GPR (1st operand) R4.

The content of the receiving field is filled with the pad character if the length of the sending field is shorter than the receiving field.

The number of bytes to be moved has to be specified, namely the length of the operands have to be specified in other registers.

The following pages explain this in detail.

Move Long (MVCL):
RR Format - Register to Register



CC Setting - 0 - Operand lengths equal.
1 - First operand length low
2 - Second Operand length high.
3 - Destructive Overlap, No Movement.

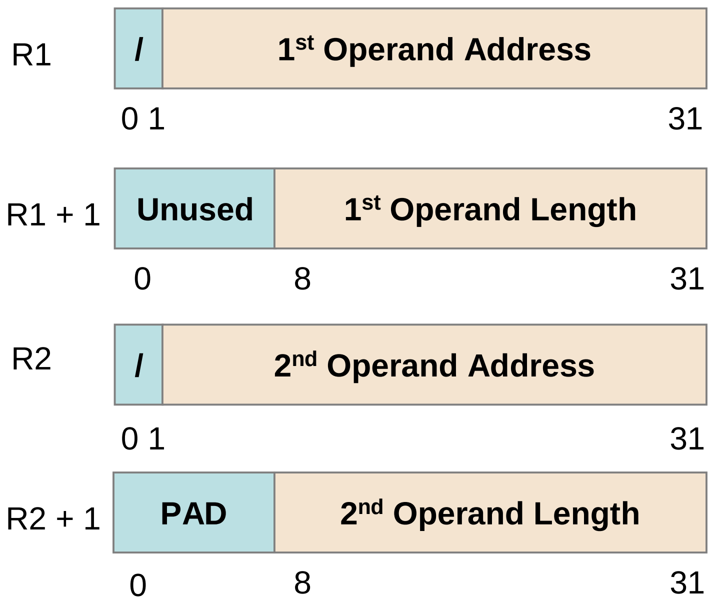
Continued…

Move Long Instruction (cont'd)

Both operands of MVCL are registers, each of which in fact implies a pair of registers. The operands must specify even numbered registers. Since each operand is a register pair, the actual operands are the even numbered registers specified, and the odd numbered registers one higher than the registers specified .The usage of the registers is as shown on the right.

The second operand is moved to the first. If the length of the second operand is shorter than the length of the first, the additional bytes on the right of the first operand are filled with the pad character.

Operands: - Registers



R1

| / | 1st Operand Address |

0 1 31

R1 + 1

| Unused | 1st Operand Length |

0 8 31

R2

| / | 2nd Operand Address |

0 1 31

R2 + 1

| PAD | 2nd Operand Length |

0 8 31

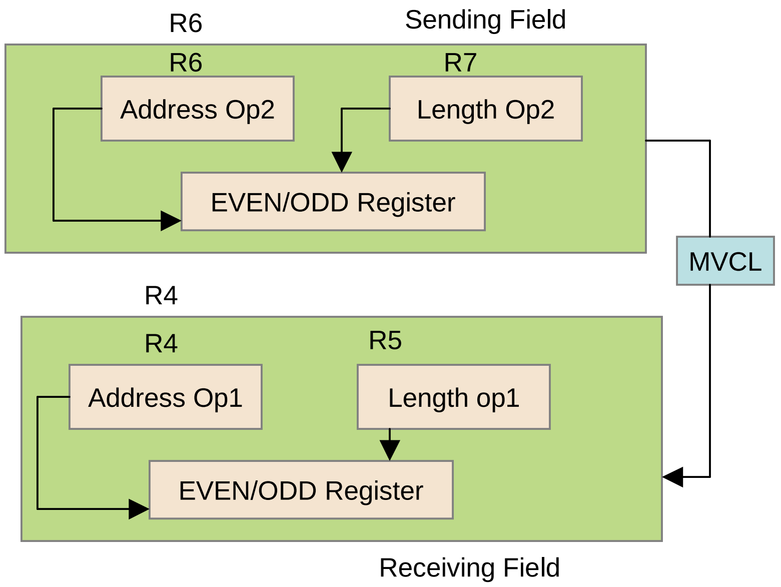
Continued…

Move Long Instruction (cont'd)

The Move Long (MVCL) instruction is best illustrated with some coding examples. In the first case, move 3000 bytes of data from a field called LONG3 to LONG4.

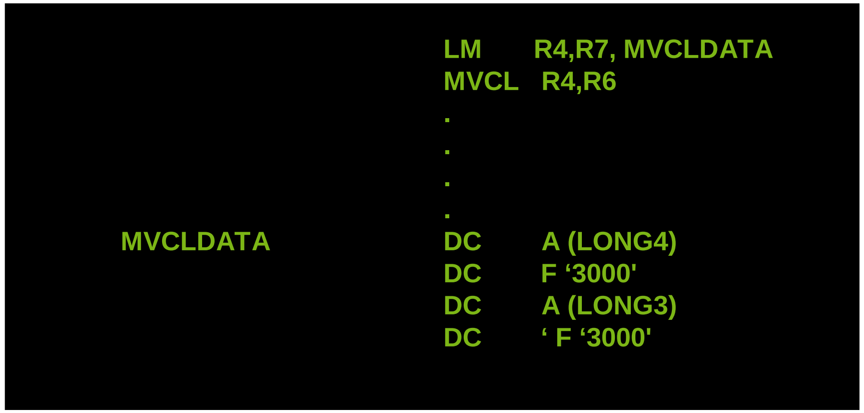
L	R4,=A(LONG4)	address of receiving field
LH	R5,=H'3000'	length of receiving field
L	R6,=A(LONG3)	address of sending field
LH	R7,=H'3000'	length of sending field

MVCL R4,R6	Move the data



Continued…

Move Long Instruction (cont’d)



```
                     LM      R4,R7, MVCLDATA
                     MVCL   R4,R6
                     .
                     .
                     .
                     .
MVCLDATA             DC      A (LONG4)
                     DC      F ‘3000'
                     DC      A (LONG3)
                     DC      ‘ F ‘3000'
```

initialize registers for MVCL to
move the data

Another way to do this, with fewer executable instructions, would be as shown above.

In this version, you set up the initial data in four consecutive full words in storage, and use LM to put it into the four registers. One thing about coding in Assembler Language is that there are usually many different ways to do the same thing.

Continued…

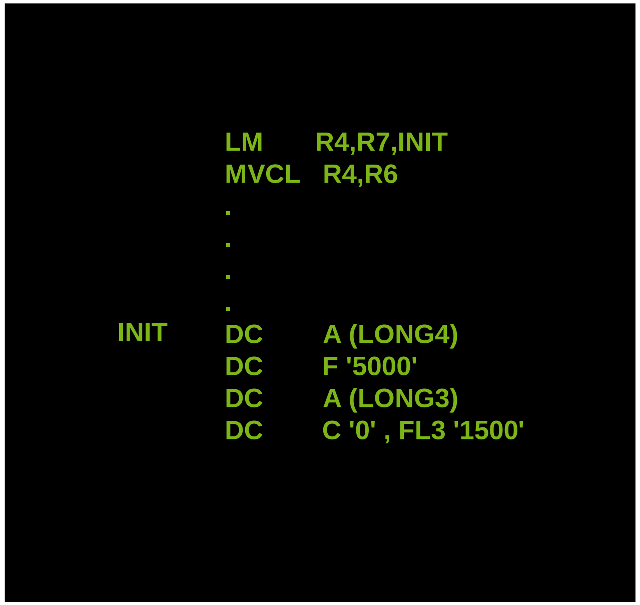
Move Long Instruction (cont’d)

For the second example of MVCL, move a field 1500 bytes long at LONG3 into a 5000 byte field at LONG4 and fill the extra 3500 bytes at the end of LONG4 with EBCDIC zeros. This is accomplished as shown on the right.

L	**R4,=A(LONG4)**	address of receiving field
LH	**R5,=H'5000'**	length of receiving field
L	**R6,=A(LONG3)**	address of sending field
LH	**R7,=H'1500'**	length of sending field
ICM	**R7,8,=C'0'**	pad character
MVCL	**R4,R6**	move data

Using LM Instruction

To set the previous example up with a LM instruction, you would use the code as shown on the right.



```
              LM      R4,R7,INIT
              MVCL   R4,R6
              .
              .
              .
              .
INIT          DC        A (LONG4)
              DC        F '5000'
              DC        A (LONG3)
              DC        C '0' , FL3 '1500'
```

Note!

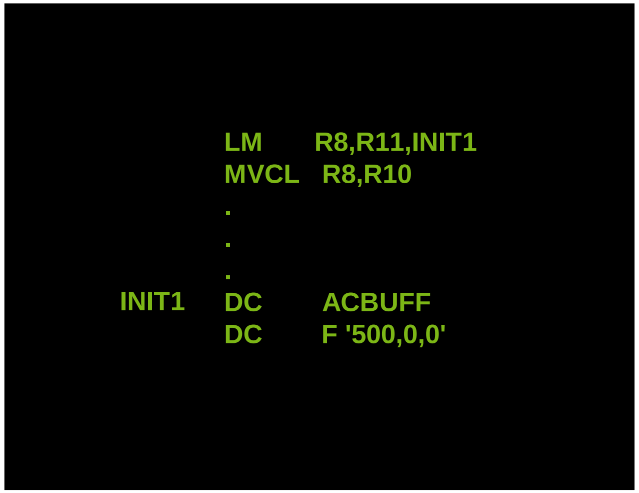
Since there is a length modifier on the FL3’1500’, no alignment is done.

Using MVCL Instruction

You can also use MVCL to initialize a field so that each byte contains a certain character, by making that character the pad character and specifying a sending field length of zero.

In this case, the address of the sending field is ignored and the whole receiving field is filled with the pad character. To fill a 500 byte field called BUFF with binary zeros, use the code shown to the right.

Here the second operand's address, length and the padding character are all zero.



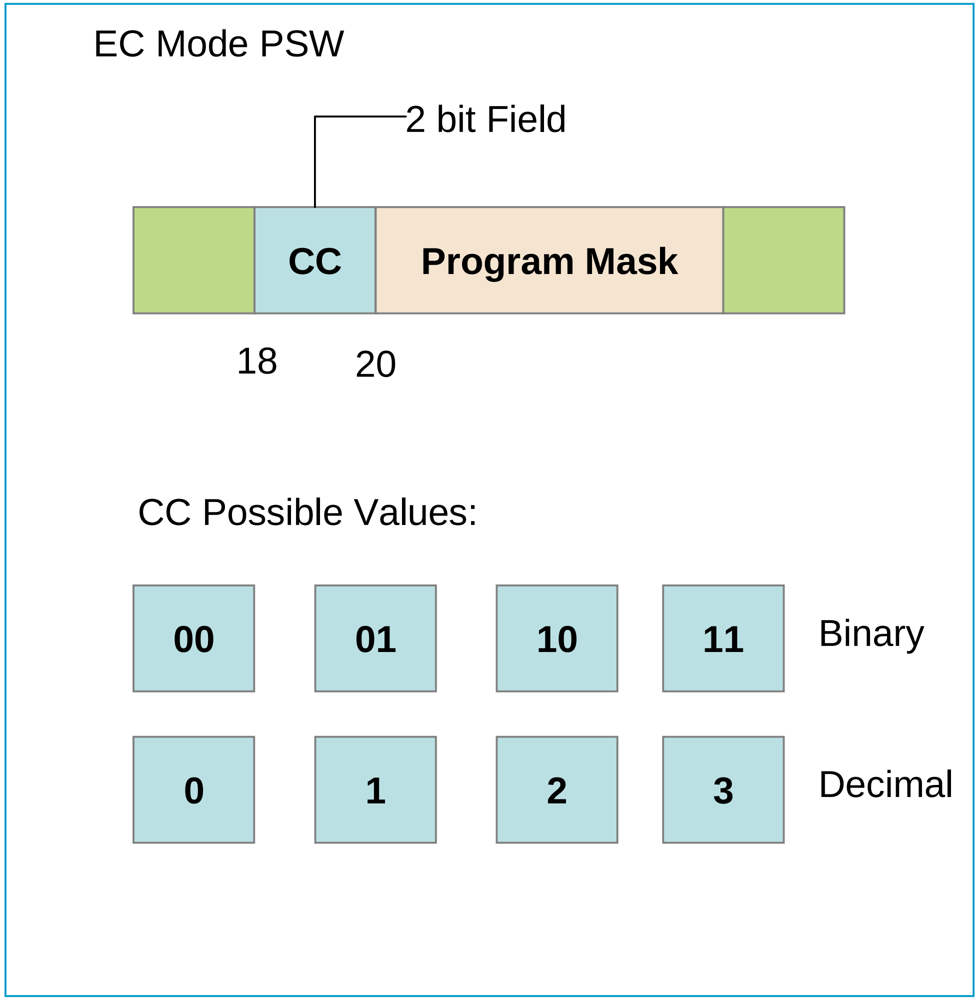
```
         LM     R8,R11,INIT1
         MVCL   R8,R10
         .
         .
         .
INIT1    DC     ACBUFF
         DC     F '500,0,0'
```

Implementing Conditional Execution

How is conditional execution implemented?

High Level Languages implement conditional execution with IF-THEN and IF-THEN-ELSE structures. They implement looping with DO-WHILE or other iterative constructs. Assembler Language does not have these high level control statements. Rather, conditional execution and looping structures must be built from lower level instructions.

In Assembler Language, conditional execution is broken into parts. One instruction tests a condition. A following instruction, a conditional branch, alters the execution path, depending on the result of the test. The condition code, a two-bit field in the Program Status Word (PSW) is where the result of the test is held, awaiting subsequent conditional branch instruction.

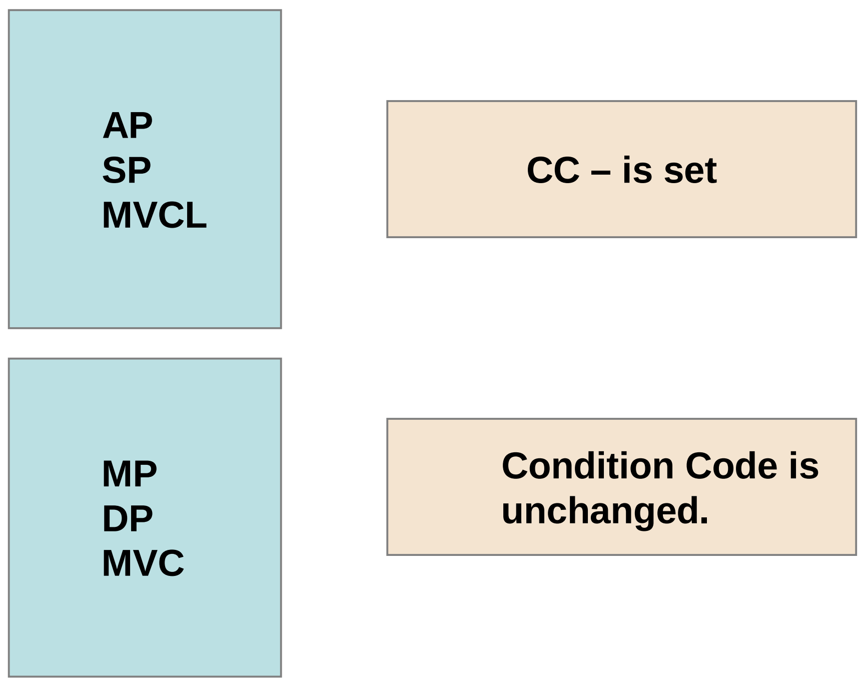


Continued…

Implementing Conditional Execution (cont'd)

Some instructions set the condition code and some do not. If and how an instruction sets the condition code is an important part of the instruction description. Once the condition code is set by an instruction, that value remains in the condition code until a subsequent instruction changes it.

Condition Code: - Some Examples



AP
SP
MVCL

CC – is set

MP
DP
MVC

Condition Code is unchanged.

Continued…

Implementing Conditional Execution (cont'd)

The condition code is a two-bit field, so that it may take on 4 unique values; 0,1,2 or 3. The CC always contains one of these four values.

You have seen that many arithmetic instructions set the condition code to indicate if the result is zero (CC=0), negative (CC=1), positive (CC=2) or if overflow has occured (CC=3). You have seen other instructions that set the condition code in other ways.

Another major group of instructions that set the condition code is the compare instructions. These instructions compare two operands and set the condition code to indicate their relative magnitudes. The condition code set by compare instructions are shown on the right.

Compare Instructions set CC as follows:

Relation of Operands	CC Setting
Op1 = Op2	0
Op1< Op2	1
Op1 > Op2	2

Continued…

Compare Instructions

A condition code setting of 3 does not occur after compare instructions. For brevity, these settings will be referred to as CC setting - Comparison in the instruction descriptions that follow.

CC Setting – Comparison:

Condition Code Setting:
0 - Equal
1 – First Operand is Low
2 – First Operand is High
3 – Not Used

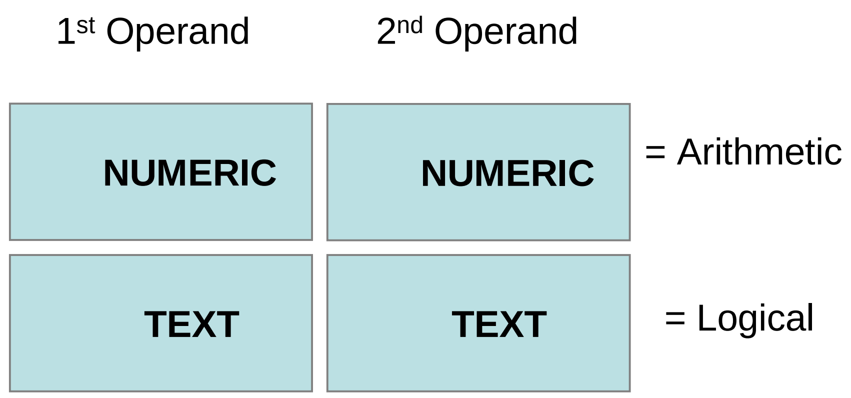
Continued…

Numeric Comparison

What is numeric comparison?

When two operands are compared, it is important that you be clear about their format. If the operands represent signed numeric quantities, then a numeric comparison is needed. On the other hand, if the operands do not represent signed numeric data, a logical comparison is performed, treating the operands simply as strings of bits.

Comparisons:



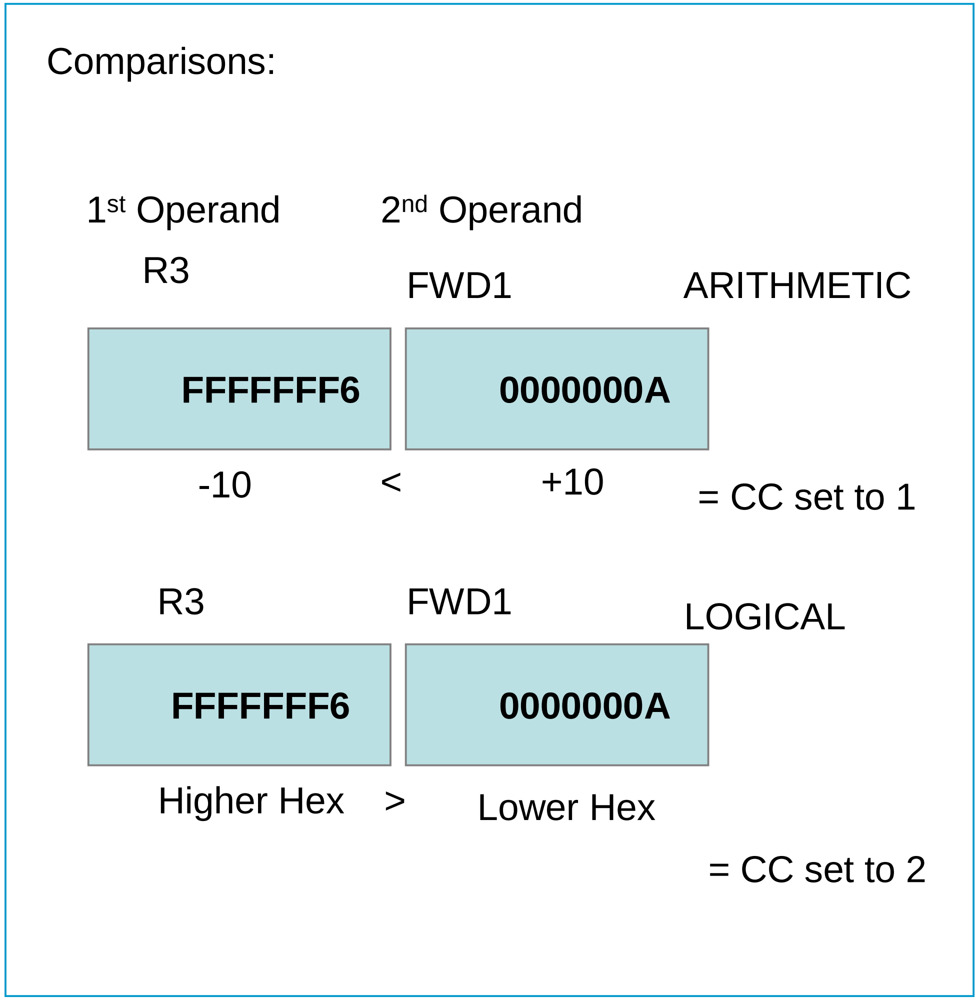
1st Operand	2nd Operand	
NUMERIC	NUMERIC	= Arithmetic
TEXT	TEXT	= Logical

Kinds of Comparison

If you compared R3 containing X'FFFFFFF6' and FDW1 containing X'0000000A', the condition code that will be set will depend on the type of comparison.

In an arithmetic comparison, the register contents would be interpreted as -10. The storage operand is interpreted as +10, so the first operand (the register) would be lower, and the condition code is set to 1.

In a logical comparison, FFFFFFF6 is a higher hex value than 0000000A, so the first operand would be higher and the condition code be set to 2.



Comparisons:

1st Operand 2nd Operand

R3 FWD1 ARITHMETIC

FFFFFFF6 0000000A

-10 < +10 = CC set to 1

R3 FWD1 LOGICAL

FFFFFFF6 0000000A

Higher Hex > Lower Hex

= CC set to 2

Logical Comparison

What is Logical Comparison?

Comparison of textual data would be a Logical Comparison, based on the relative position of the characters in the EBCDIC code tables. If C'A' is compared with C'5', the second operand would be larger, since the EBCDIC code for 5 (F5) is higher than the EBCDIC code for A (C1), as shown on the table in the right.

EBCDIC CODE	
A	C1
5	F5

Compare Instruction

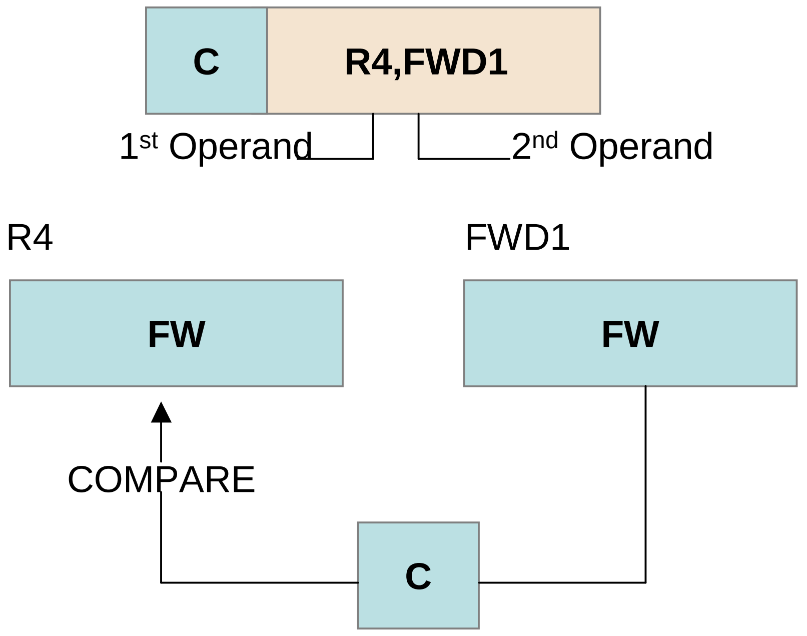
What is the Compare (C) Instruction?

The Compare (C) instruction is used to arithmetically compare the first operand, a fullword in GPR and the second operand, a fullword in main storage. Neither operand is modified but the condition code is set to indicate the result of the comparison.

For example: C R4,FWD1

The contents of R4, as shown in the table on the right, are compared arithmetically to the contents of the fullword at location FWD1. The condition code is set to indicate the result of the comparison.

Compare (C):
RX Format – Register to Indexed Storage



CC Setting:
If R4 = FWD1 -- 1
If R4 < FWD1 -- 1
If R4 > FWD1 -- 2

Compare Halfword Instruction

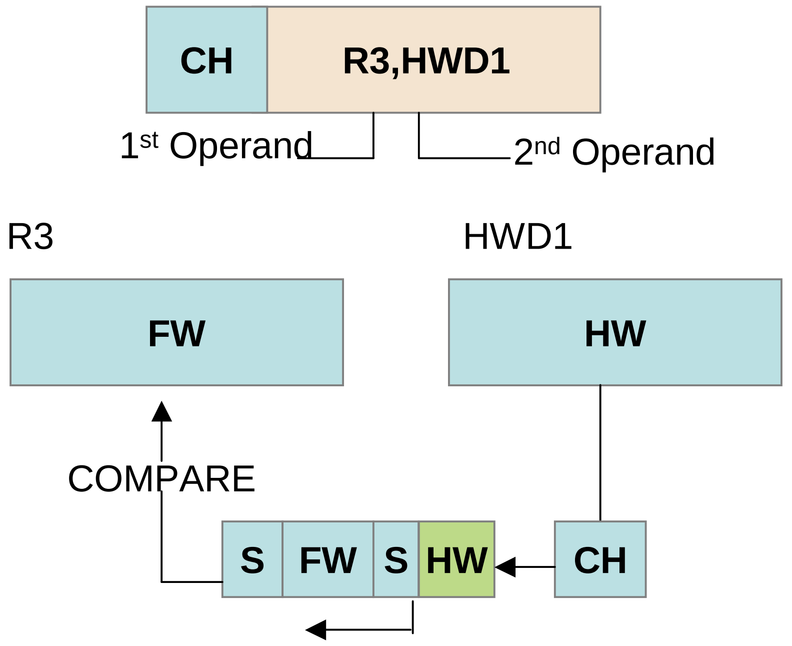
What is the Compare Halfword (CH) Instruction?

The Compare Halfword (CH) instruction is used to expand the second operand, a halfword in main storage, to a fullword by sign extension, and compare it arithmetically to the first operand, a fullword in a GPR. Neither operand is modified but the condition code is set to indicate the result of the comparison.

For example: CH   R3, HWD1

The halfword at HWD1, as shown in the table on the right, is transparently expanded to a fullword through sign extension, and compared arithmetically to the value in R3. The condition code is set to indicate the result of the comparison.

Compare Half word (CH):
RX Format – Register to Indexed Storage



CC Setting: SIGN EXTENSION
If R3 = HWD1 -- 0
If R3 < HWD1 -- 1
If R3 > HWD1 -- 2

Compare Register Instruction

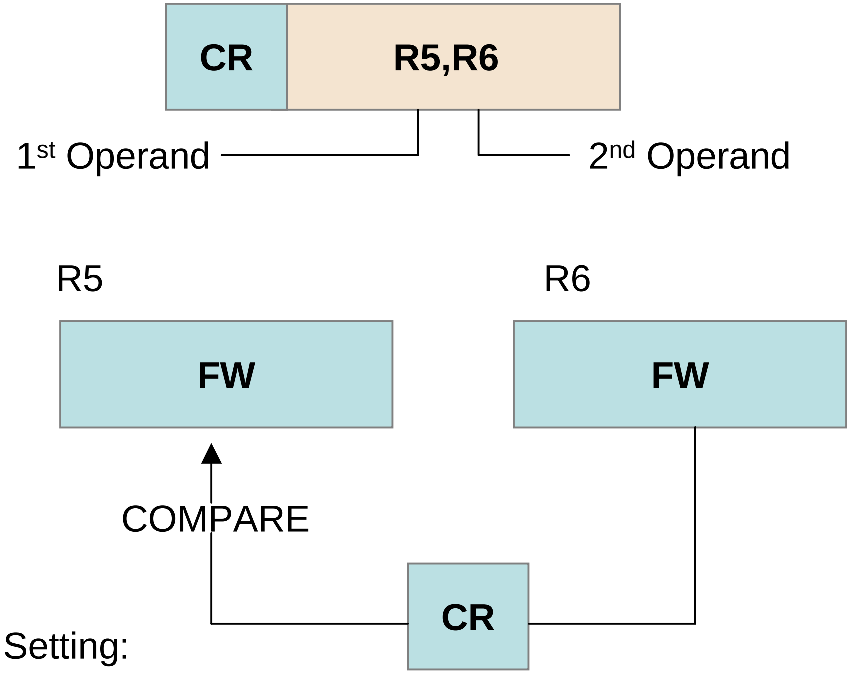
What is the Compare Register (CR) Instruction?

The Compare Register (CR) instruction is used to compare two operands, both fullwords in GPRs, arithmetically. Neither operand is modified but the condition code is set to indicate the result of the comparison.

For example: CR R5,R6

The contents of R5 are compared arithmetically to the contents of R6, as shown in the table on the right. The condition code is set to indicate the result of the comparison.

Compare Register (CR):
RR Format – Register to Register

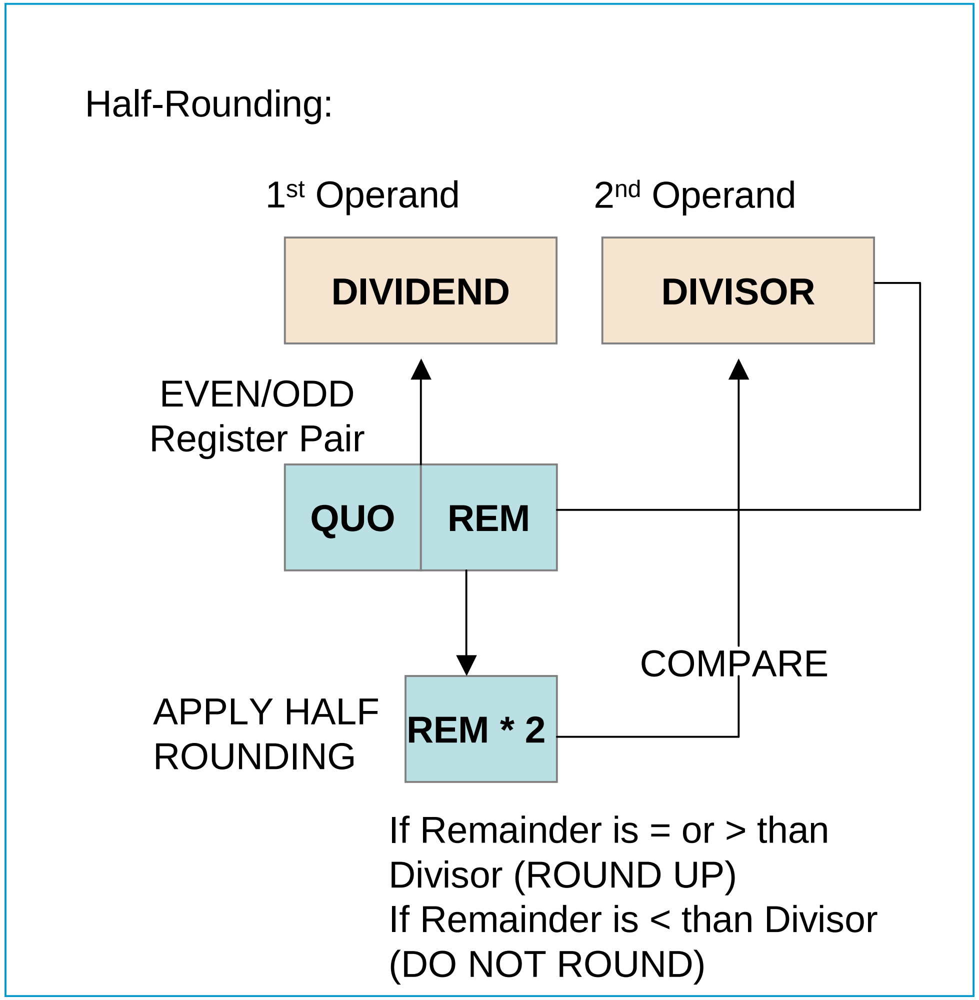


CC Setting:
If R5 = R6 -- 0
If R5 > R6 -- 1
If R5 > R6 -- 2

Fixed-Point Compare Instruction

Fixed-point compare instructions are used whenever the larger of the two fixed-point quantities is to be determined, or if they are equal. One common application is rounding after division.

Fixed-point division yields a fixed-point quotient and a fixed-point remainder. In many applications, you will want to round the quotient up to the next higher integer if the remainder is equal to or greater than half the divisor. This is called half rounding.



Compare Decimal Instruction

What is the Compare Decimal (CP) instruction?

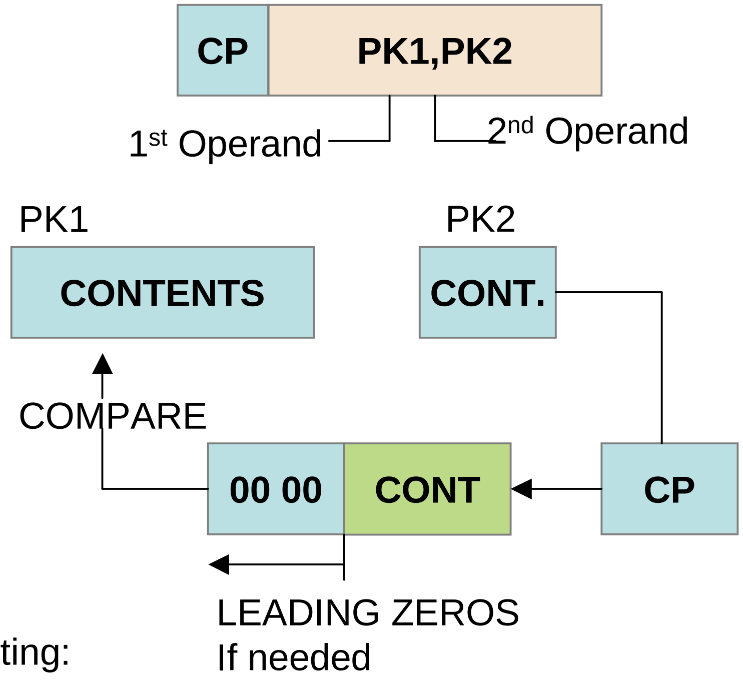
Packed decimal operands too have an arithmetic comparison operation.

The Compare decimal (CP) instruction is used to compare two operands, both packed numbers in main storage arithmetically. If the fields are of different lengths, the shorter length is expanded by inserting leading zeroes. Neither operand is modified, but the condition code is set to indicate the result of the comparison.

For example: CP   PK1,PK2

As shown in the example on the right, the contents of PK1 and PK2 are compared arithmetically. The condition code is set to indicate the result of the comparison.

Compare Decimal (CP):
SS Format – (2 length) Storage to Storage



CC Setting:

If PK1 = PK2   --   0
If PK1 < PK2   --   1
If PK1 > PK2   --   2

Unit: Comparing and Branching   Topic: Compare Instructions

Rounding Up – Example 1

```
               ZAP      QUOREM, SUM        move sum work area
               DP       QUOREM, COUNT      divide to get quotient and remainder
               AP       REM,  REM          double the remainder
               CP       REM, COUNT          is it less then divisor?
NOROUND        BL       NOROUND            Yes – don’t round
               AP       QUO, = P ‘1’        no – round up
SUM            EQU       *
COUNT   DS               .
QUOREM         DS       PL6
QUO            DS       PL3
REM            DS       PL9
               DS       PL6
               DS       PL3
```

You may want to implement half rounding with decimal arithmetic. Suppose a series of numbers have been summed into SUM, and the numbers have been counted into a field called COUNT. The average of the numbers, half rounded would be calculated as shown above.

Rounding Up – Example 2

```
                ZAP     CALC,  MARK       move mark to calculate field
                MP      CALC, = P ’11’     multiply by 1.1 to add 10% (10 times too large)
                SRP     CALC, 64 –1,5      divide by 10 and round
                CP      CALC, = P’100’     is adjusted mark  > 100?
                BNH     NOADJUST           no (branch if first operand not high)
                ZAP     CALC, = P’100’     yes – reduce to 100
NOADJUST        EQU     *                  move increased mark back to mark
                ZAP     MARK, CALC
MARK            .
CALC            .

MARK            DS  PL2
CALC            DS  PL2
```

In this example, a teacher wants to raise student marks on a test by 10%, but also wants to assure that no mark exceeds 100. If the original mark was in a field called MARK, the above code would perform the calculation for a single student.

Compare Logical Register

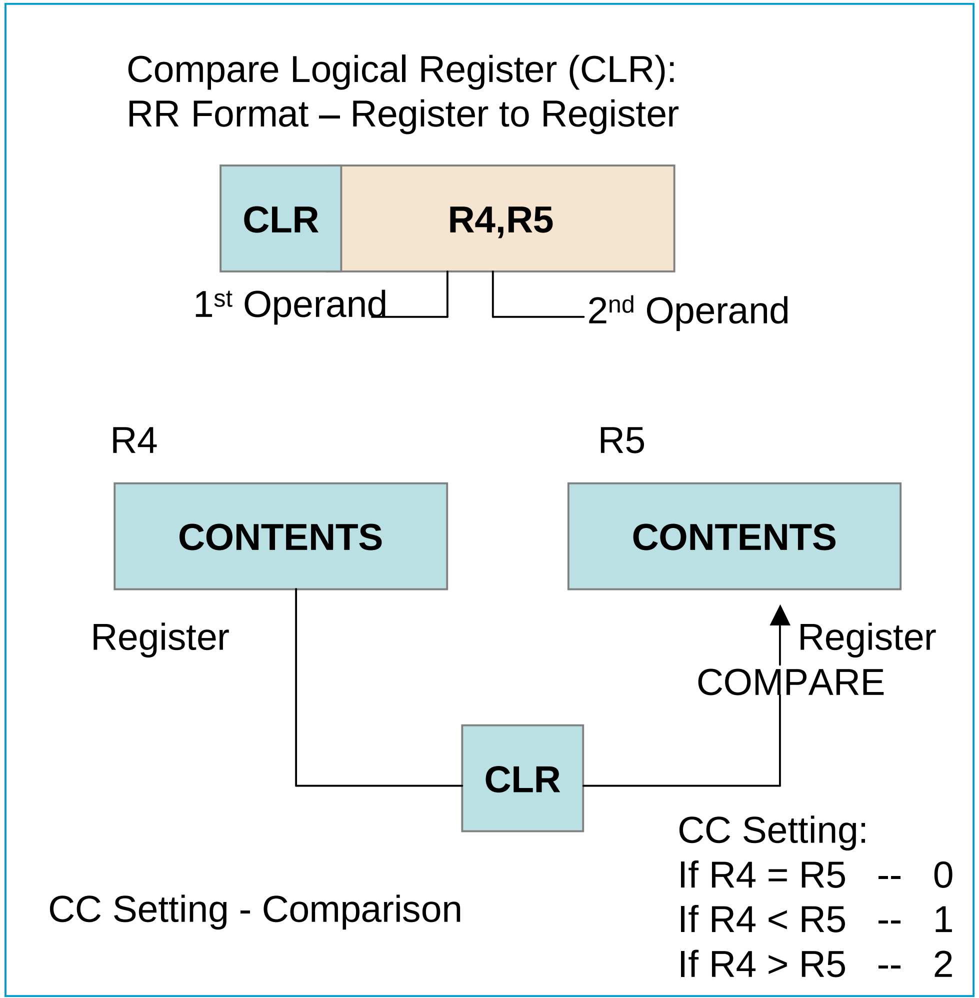
What is the Compare Logical Register?

There are several logical comparison instructions to handle the various types of operands. In each case, the instruction compares its two operands, treating them as logical quantities that are simply strings of bits.

The Compare Logical Register (CLR) instruction is used to compare its operands logically, both 32 bit logical quantities in GPRs. Neither operand is modified but the condition code is set to indicate the result of the comparison.

For example: CLR R4,R5

The contents of R4 and the contents of R5 are compared logically. The condition code is set to indicate the result of the comparison.



CC Setting:
If R4 = R5 -- 0
If R4 < R5 -- 1
If R4 > R5 -- 2

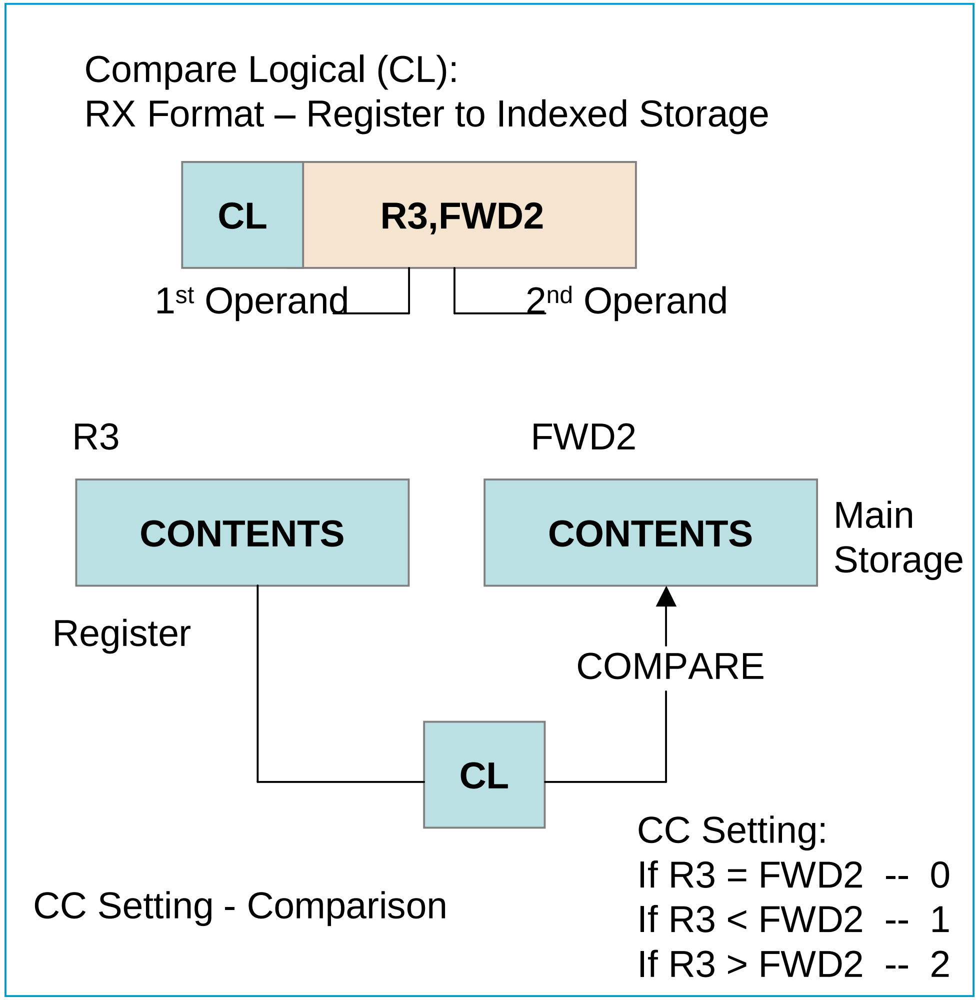
Compare Logical Instruction

What is the Compare Logical Instruction?

The Compare Logical (CL) instruction is used to compare logically the first operand, a 32 bit logical quantity in a register, with the second operand, a 32 bit logical quantity in main storage. Neither operand is modified but the condition code is set to indicate the result of the comparison.

For example: CL R3,FWD2

The contents of R3 and the contents of FWD2 are compared logically. The condition code is set to indicate the result of the comparison.



Compare Logical Immediate Instruction

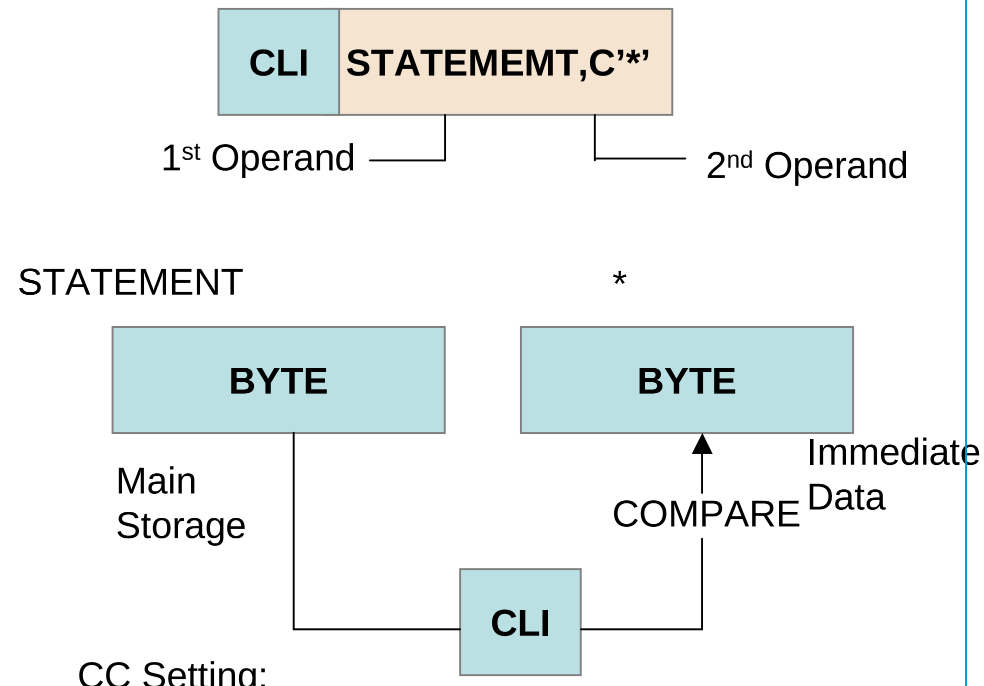
What is the Compare Logical Immediate Instruction?

The CLI instruction is used to compare the first operand logically with the second operand. The first operand is a byte in main storage. The second operand is a byte of immediate data in the instructions. Neither operand is modified during comparison but the condition code is set to indicate the result of the comparison.

For example: CLI STATEMENT,C'*'

The byte of main storage at STATEMENT is compared to an asterisk. The condition code is set to indicate the result of the comparison.

Compare Logical Immediate (CLI):
SI Format – Storage to Immediate Data



CC Setting:
If STATEMENT = C’*’ -- 0
If STATEMENT < C’*’ -- 1
If STATEMENT > C’*’ -- 2

Continued…

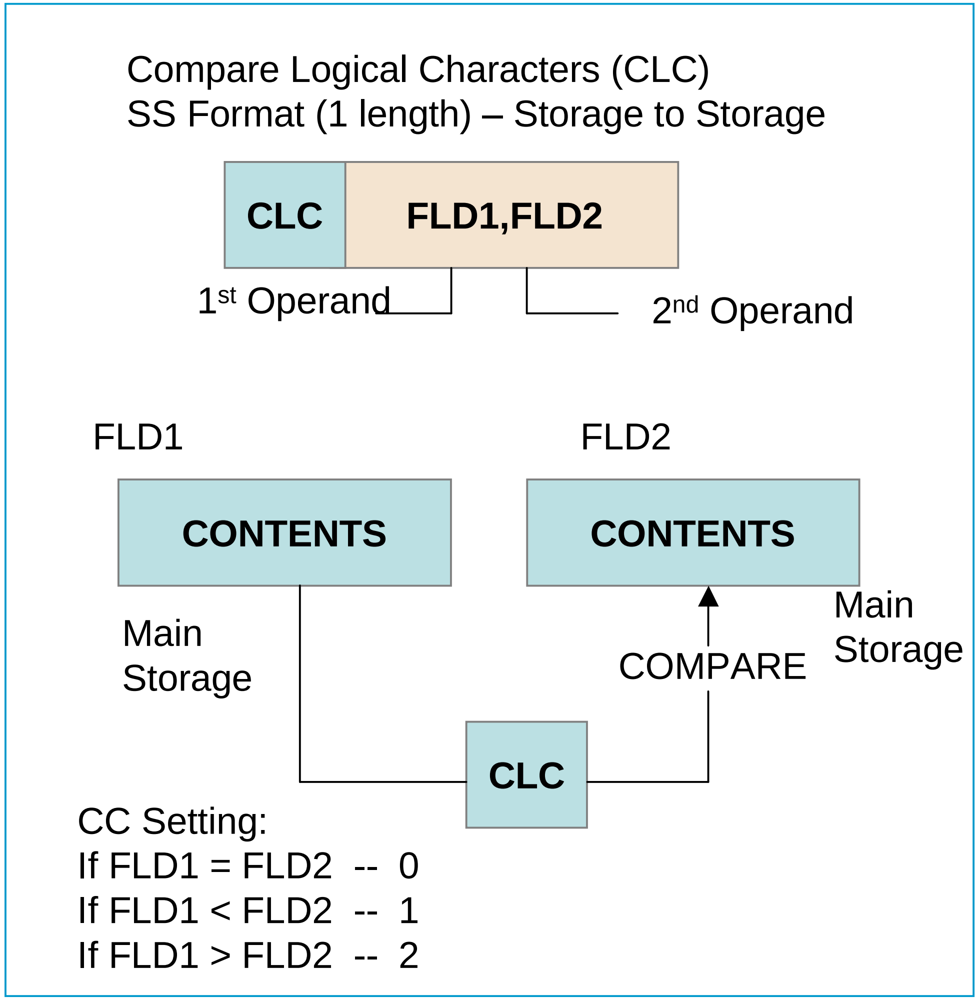
Compare Logical Characters

What is the Compare Logical Character?

The Compare Logical Characters (CLC) instruction is used to compare logically the two operands, equal length character strings in storage. Neither operand is changed, but the condition code is set to indicate the result of the comparison.

For example: CLC   FLD1,FLD2

The contents of FLD1 and FLD2 are compared logically. The condition code is set to indicate the result of the comparison.



Continued…

Compare Logical Characters Under Mask Instruction

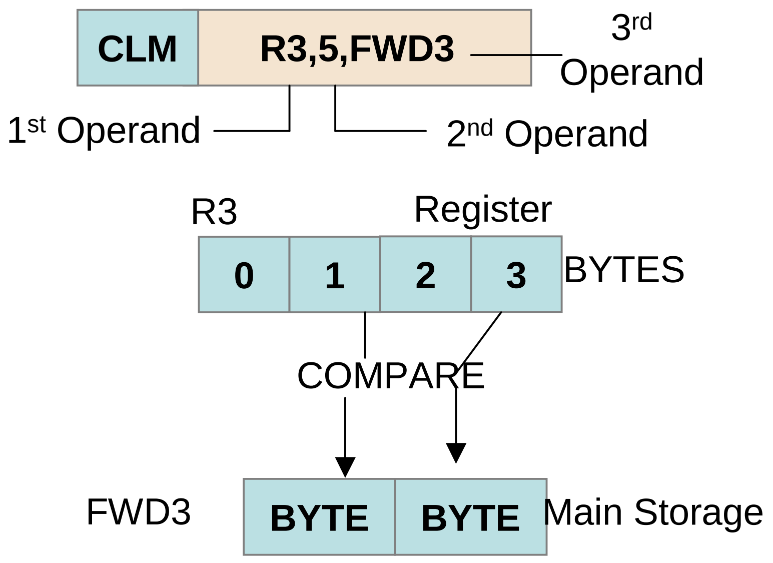
What is the CLM instruction?

The Compare Logical Characters Under Mask (CLM) instruction is used to compare logically the bytes of the GPR specified by 1 bit in the mask, with an equal number of consecutive bytes in main storage beginning at the third operand location. The operands are unchanged, but the condition code is set to indicate the results of the comparison.

For example: CLM R3,5,FWD3

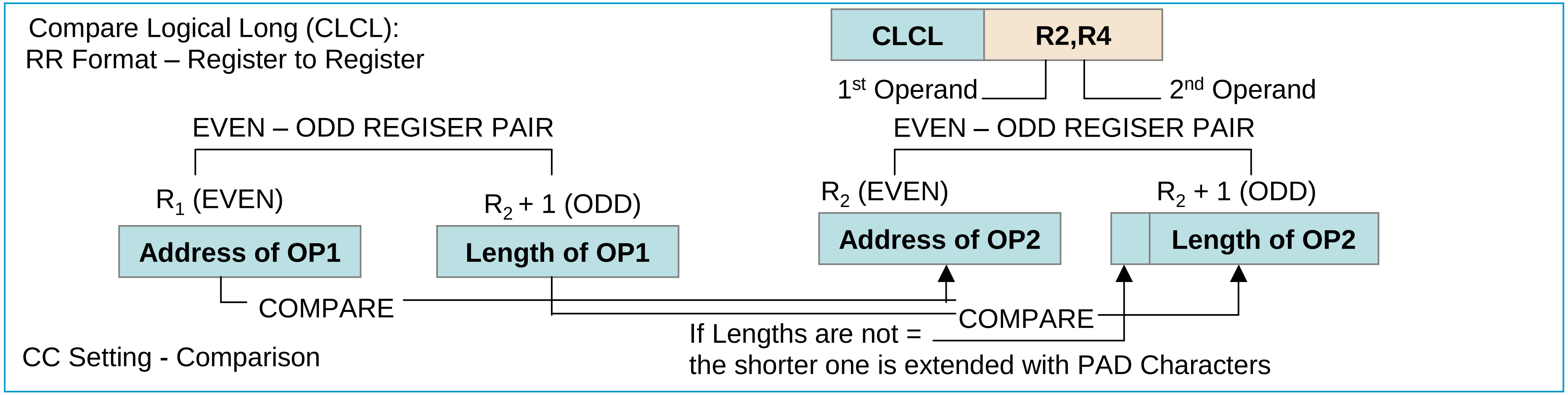
Bytes 1 and 3 of R3 are compared logically with the two bytes in storage beginning at FWD3. The condition code is set to indicate the results of the comparison.

Compare Logical Characters Under Mask (CLM):
RS Format – Register to Storage



CC Setting:
If R3 bytes = FWD3 -- 0
If R3 bytes < FWD3 -- 1
If R3 bytes > FWD3 -- 2

Compare Logical Characters Long Instruction



This instruction is analogous to the Move Long instruction (MVCL). Like Move Long, both operands must specify even numbered GPRs using the Compare Logical Long (CLCL) instruction, which, in each case represent even-odd register pairs. The even numbered registers contain the addresses of the two operands. The odd numbered registers contain the length of the operands, in their low order 24 bits. The high order byte of R_2+1 contains the padding byte.

Continued…

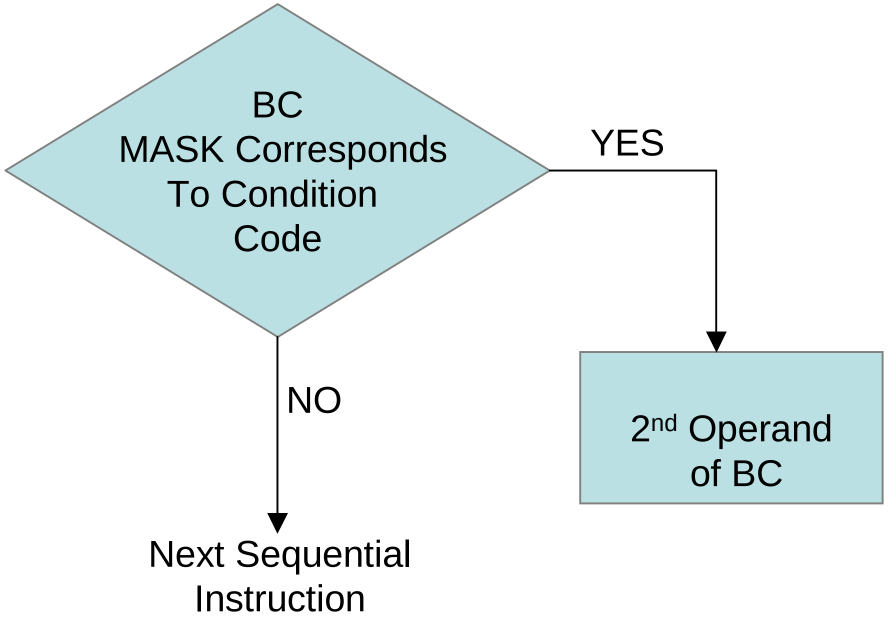
Branch on Condition Instruction

What is the Branch on Condition Instruction?

The way you implement conditional execution in Assembler Language is with the Branch on Condition (BC) Instruction. This instruction tests the current setting of the condition code, using a mask field in the instruction.

If the mask field corresponds to the setting of the condition code, then the next instruction to be executed is the one at the address specified as the second operand of Branch on Condition. If there is no correspondence, then execution proceeds sequentially with the instruction following Branch on Condition.

Branch on Condition



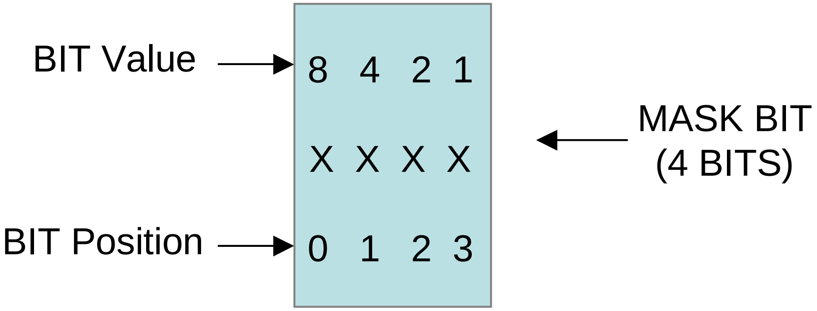
The Mask Field

The mask field is 4 bits in length, allowing 16 possible values. The condition code is 2 bits long, allowing 4 possible values. The 4 bits of the mask correspond to the 4 possible condition values.

If the mask value is 8, the Branch on Condition instruction tests for a CC setting of zero. If the CC = 0, the branch is taken, otherwise normal sequential execution occurs.

Mask Field:

Condition Code Setting	0	1	2	3
Corresponding Mask Bit	0	1	2	3
Mask Value	8	4	2	1



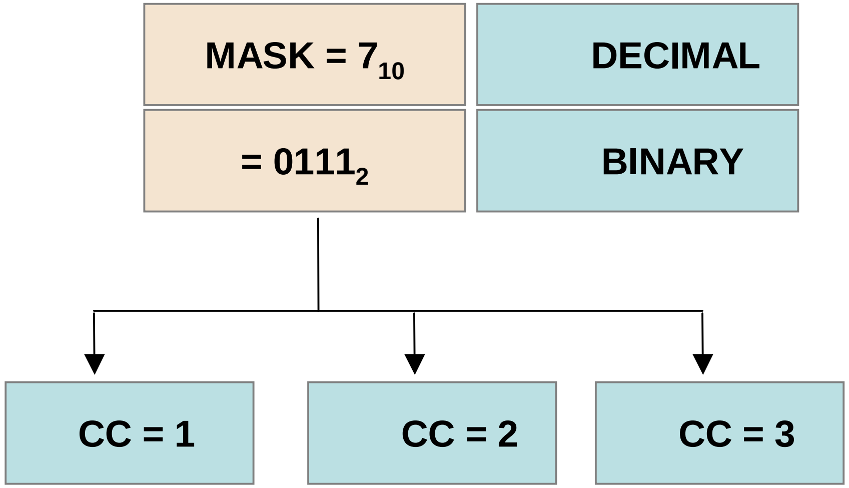
Continued…

The Mask Field (cont'd)

Mask values can be any value in the range of 0 to 15. A mask value of 7 (4+2+1) tests if the CC is 1 or 2 or 3. If so, the branch is taken. Otherwise it is not.

Any combination of condition codes can be tested in a single Branch on Condition instruction. A mask value of 15 tests if the CC is 0 or 1 or 2 or 3. Since the CC must be one of these values, this constitutes an unconditional branch.

Mask Values:



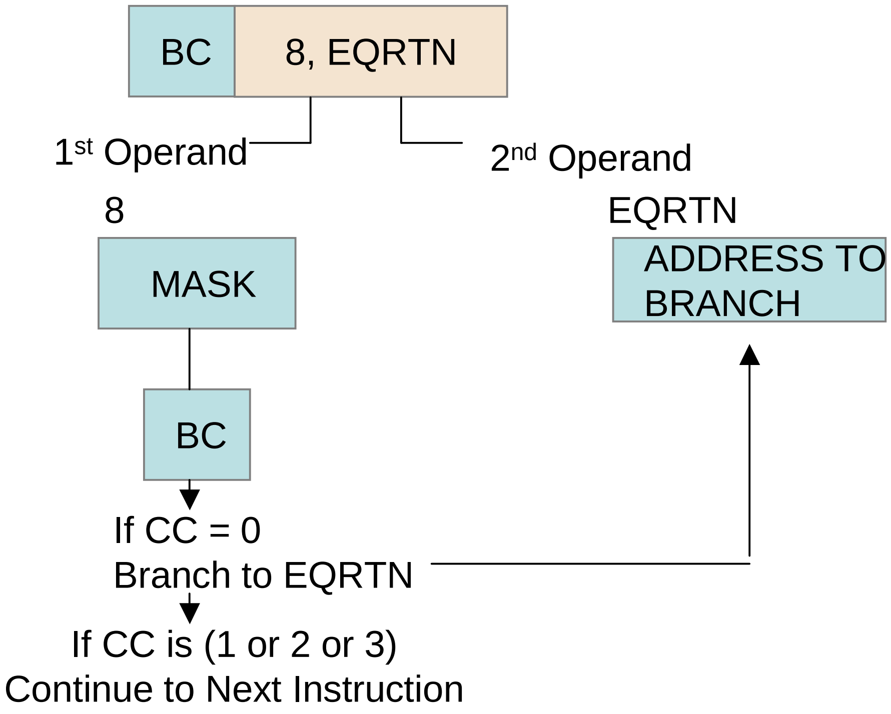
BC – An Example

When using the Branch on Condition (BC) instruction the first operand of BC specifies a mask, not a GPR. If the bit in the mask corresponding to the current value of the condition code is on, then the address specified as the second operand replaces the next instruction address in the PSW. If not, the normal sequential instruction sequencing occurs.

For example: BC 8,EQRTN

If the condition code is zero, branch to EQRTN. Otherwise, continue with the next instruction after the BC.

Branch on Condition (BC):
RX Format – Register to Indexed Storage



CC Setting – Condition Code is Unchanged

Continued…

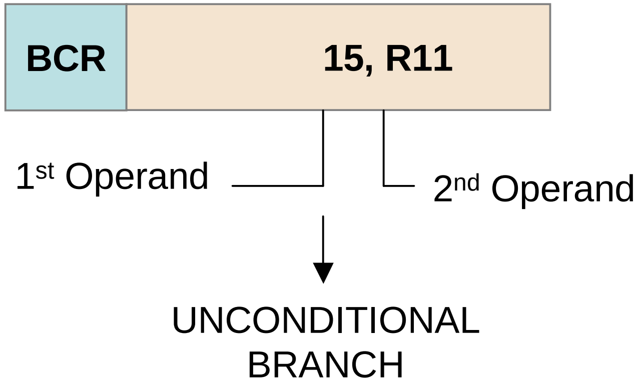
Branch on Condition Register Instruction

When using the Branch on Condition Register (BCR) instruction, the first operand specifies a mask, not a GPR. If the bit in the mask corresponding to the current value of the condition code is on, then the address in the second operand register replaces the next instruction address in the PSW. If not, normal instruction sequencing occurs. However, if the second operand specifies 0, no branch is taken regardless of the mask and condition code values.

For example: BCR 15,R11

Regardless of the condition code setting, branch to the address in register 11.

Branch on Condition Register (BCR):
RR Format – Register to Register



CC Setting – Condition Code is Unchanged

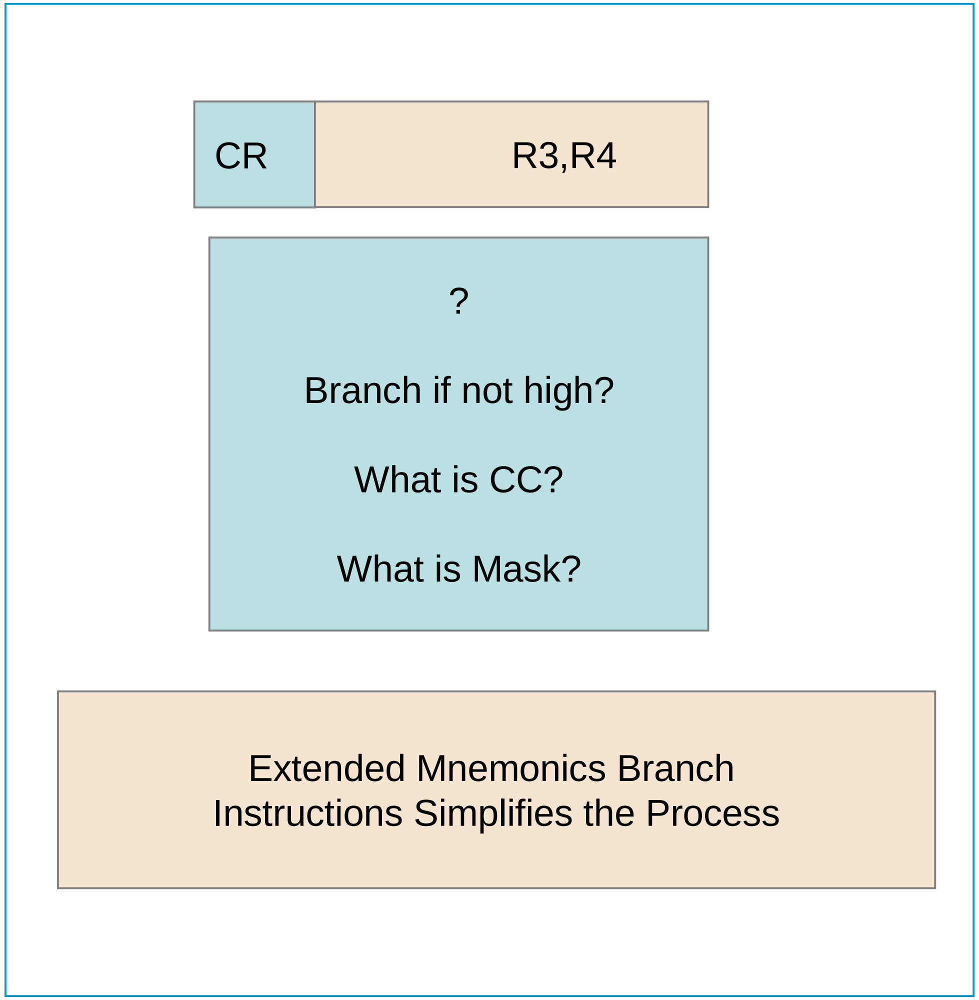
Extended Mnemonic Branch Instructions

What are Extended Mnemonic branch instructions?

The two operands of the Branch on Condition instruction specify the conditions under which a branch is taken, and the branch address.

It can be hard to remember the actual condition code for the various instructions and to calculate mask values, particularly when testing combination of condition codes.

Therefore, the Assembler Program simplifies the process by introducing extended mnemonic branch instructions.


CR
R3,R4
?
Branch if not high?
What is CC?
What is Mask?
Extended Mnemonics Branch Instructions Simplifies the Process

Continued…

Extended Mnemonic Branch Instructions (cont'd)

The extended mnemonic branch instructions combine the operation code for branching and the mask, which specifies the conditions under which the branch is taken, into a simple extended mnemonic.

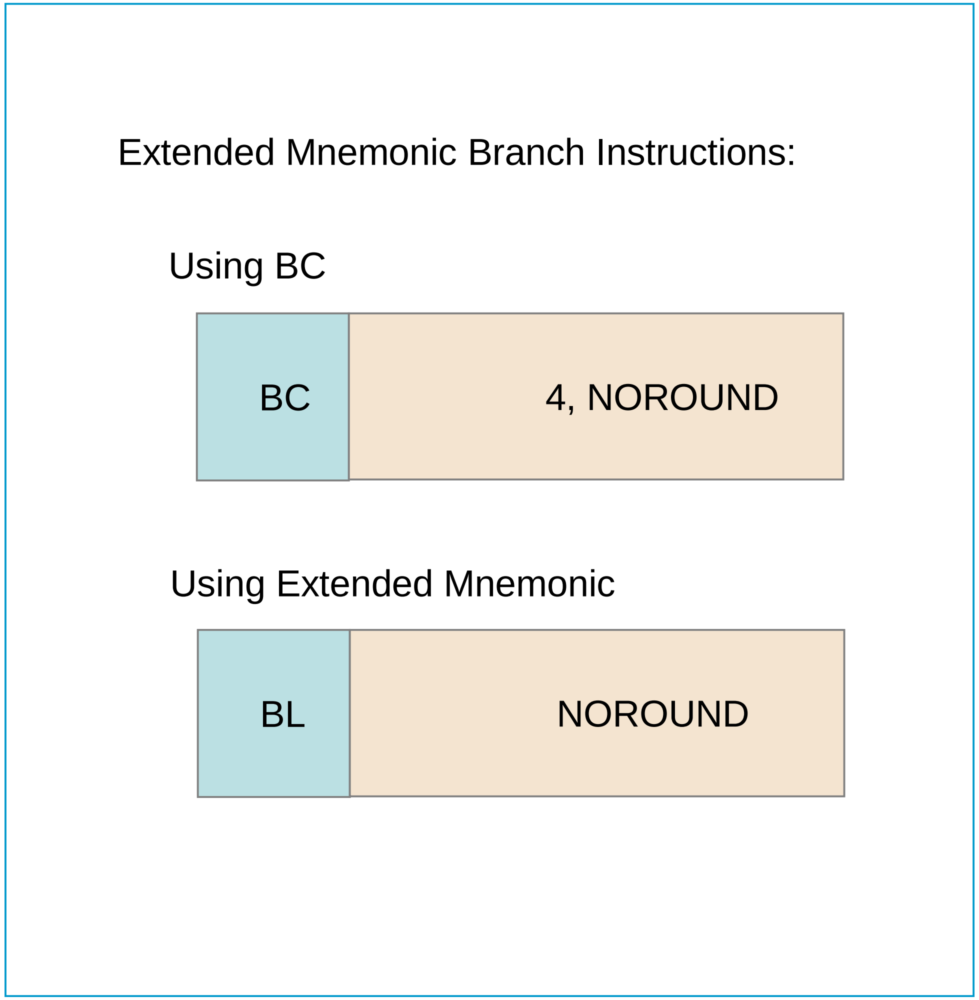
For example, if you wanted to branch to NOROUND if the value in R3 was less than the value in R4, you would specify:

```
CR   R3,R4
BC   4,NOROUND
```

With extended mnemonics, the code would be as follows:

```
CR  R3,R4
BL  NOROUND
```

The extended mnemonic BL (branch if first operand low) takes the place of the actual instruction BC and the mask value of 4.



Extended Mnemonic Branch Instructions:

Using BC

Using Extended Mnemonic

Continued…

Extended Mnemonics (Logical)

Extended Mnemonics Branch Instructions: (Logical)

Mask Value	RX Extended Mnemonic	RR Extended Mnemonic	Interpretation
2	BH	BHR	Branch if Op1 High
4	BL	BLR	Branch if Op1 Low
7	BNE	BNER	Branch if Not Equal
8	BE	BER	Branch if Equal
11	BNL	BNLR	Branch if Op1 Not Low
13	BNH	BNHR	Branch if Op1 Not High

There is a set of extended mnemonics you can use after comparison instructions. They are shown above.

Extended Mnemonics (Arithmetic)

Extended Mnemonics Branch Instructions: (Arithmetic)

Mask Value	RX Extended Mnemonic	RR Extended Mnemonic	Interpretation
1	BO	BOR	Branch on Overflow
2	BP	BPR	Branch on Plus
4	BM	BMR	Branch on Minus
7	BNZ	BNZR	Branch on Not Zero
8	BZ	BZR	Branch on Zero
11	BNM	BNMR	Branch on Not Minus
13	BNP	BNPR	Branch on Not Plus
14	BNO	BNOR	Branch on No Overflow

After arithmetic instructions, you use the extended mnemonics shown above.

Test under Mask Instruction

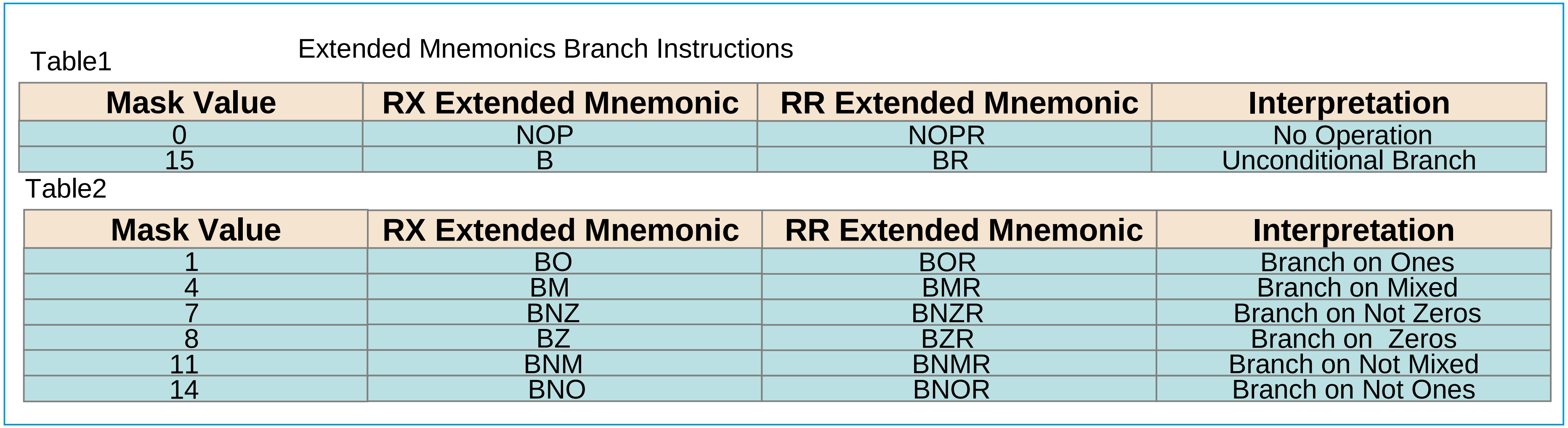


Table1 Extended Mnemonics Branch Instructions

Mask Value	RX Extended Mnemonic	RR Extended Mnemonic	Interpretation
0	NOP	NOPR	No Operation
15	B	BR	Unconditional Branch

Table2

Mask Value	RX Extended Mnemonic	RR Extended Mnemonic	Interpretation
1	BO	BOR	Branch on Ones
4	BM	BMR	Branch on Mixed
7	BNZ	BNZR	Branch on Not Zeros
8	BZ	BZR	Branch on Zeros
11	BNM	BNMR	Branch on Not Mixed
14	BNO	BNOR	Branch on Not Ones

The general extended Mnemonics are shown in Table 1 above.

The Test under Mask instruction that is used to test selected bits in a byte, has its own set of extended mnemonics, as shown in Table 2. This instruction is discussed in the Assembler Language (Advanced) Course.

The use of extended branch mnemonics makes a program easier to read, as well as easier for the programmer to code.

IF-THEN Structure in C

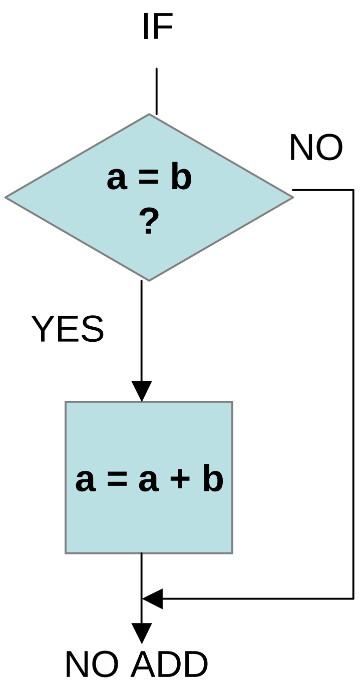
A basic IF-THEN structure in C looks like this:

```
if (a=b) {
        a:=a+b;
         }
```

The corresponding code in Assembler Language would look like this:

```
        CP    A,B          if (a=b) {
        BNE   NOADD
        AP    A,B
NOADD   EQU   *            }
```

IF – THEN Structure:



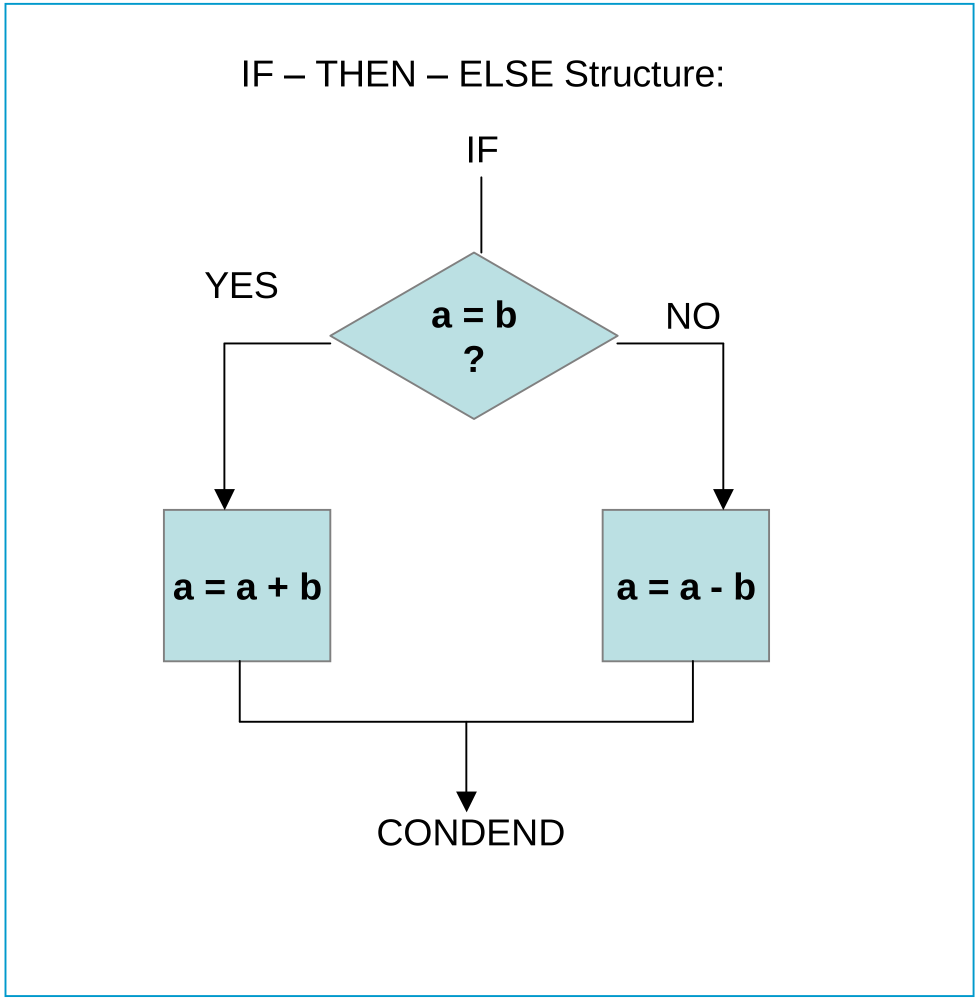
IF-THEN-ELSE Structure

A basic IF-THEN-ELSE structure in C looks like this:

```
if(a=b) {
        a:=a+b;
         }
else    {
     a:=a-b;
         }
```

The corresponding Assembler Language code would be:

```
                  CP       A,B              if (a=b) {
                  BNE      SUBTR
                  AP       A,B
                  B        CONDEND
SUBTR             EQU      *                } else {
                  SP       A,B
CONDEND           EQU      *                }
```



DO – WHILE Structure

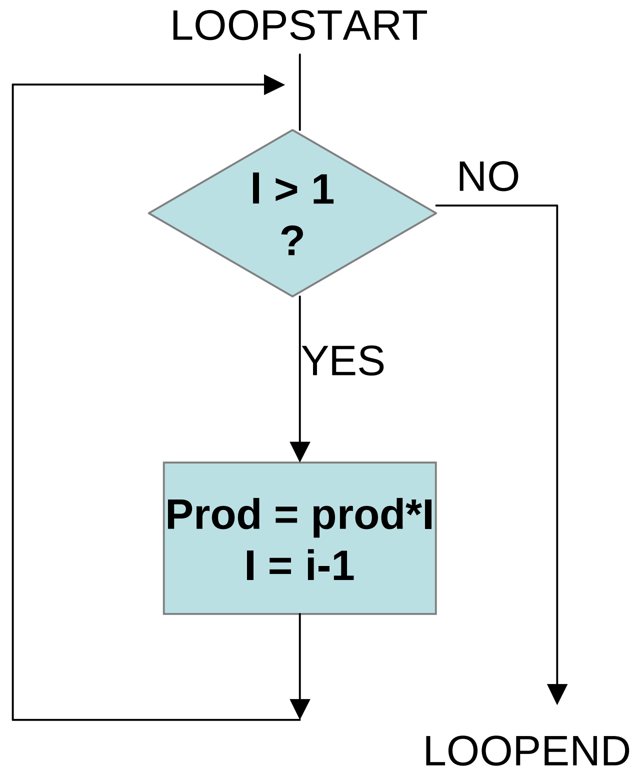
The most common and flexible looping structure is DO-WHILE. Like the conditional structures, it must be built in Assembler Language using the Branch on Condition statement. A DO – WHILE loop in C might look like this:

```
while (i>1){
          prod:=prod*i;
          i=i-1;
          }
```

The equivalent Assembler Language code would look like this:

```
LOOPSTART     EQU     *
              CP      I,=P'1'   while (I >1) {
              BNH     LOOPEND
              MP      PROD,I
              SP      I,=P'1'
              B       LOOPSTART
LOOPEND       EQU     *              }
```

DO – WHILE Structure:



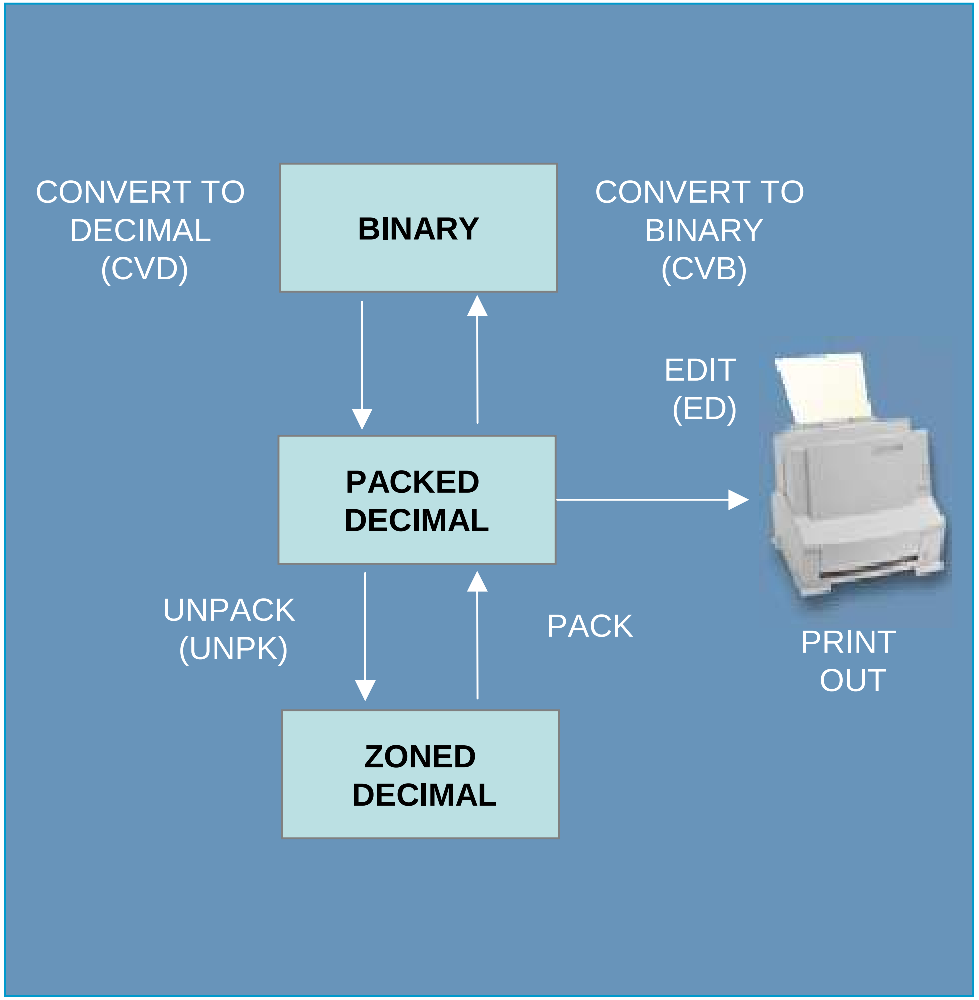
Storing Numeric Data

How is numeric data read and stored?

Numeric data can be read and stored in a computer in three formats:

- Binary
- Packed Decimal
- Zoned decimal

This unit will introduce you to the instructions that work with conversions, which are Convert to Binary (CVB) and Convert to Decimal (CVD).



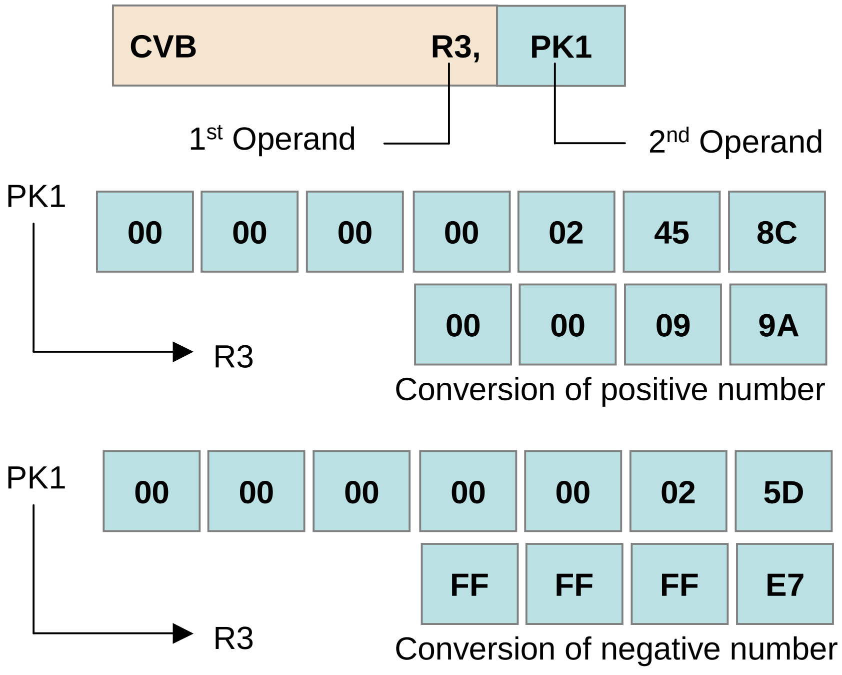
Convert to Binary Instruction

What is Convert to Binary instruction?

The Convert to Binary (CVB) instruction converts a number from the packed decimal to fixed-point integer form. Conversion of data into fixed-point format requires that it first be in an 8 byte packed decimal field.

System /370 and earlier architectures required that it be aligned on a doubleword boundary. If the second operand is not a valid 8-byte packed number, then a data exception will occur. If the value being converted is too large to fit into a GPR, a fixed-point divide exception will occur.

Convert to Binary (CVB):
RX Format - Register to indexed storage



Twos complement form of negative number
CC Settings: Condition Code is Unchanged

Continued…

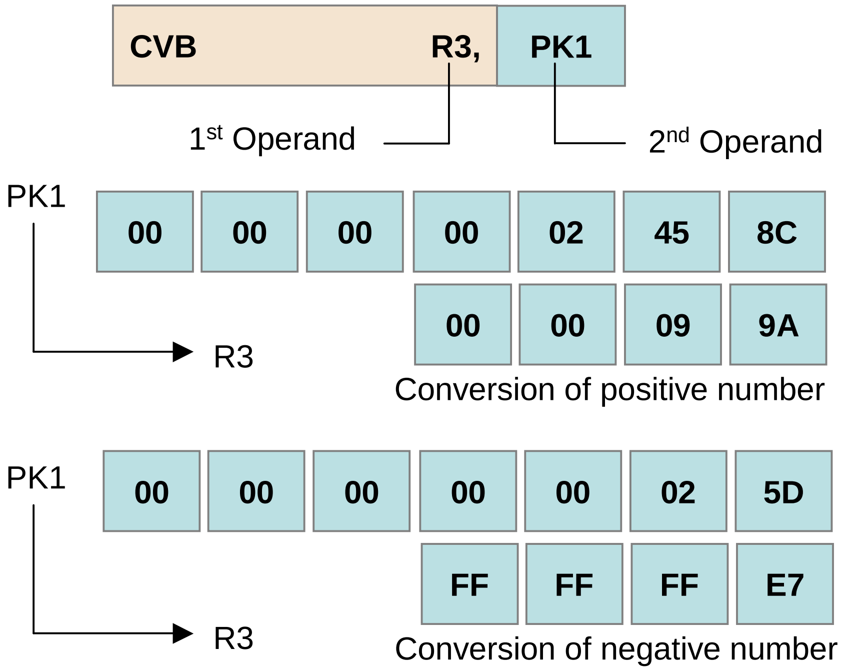
Convert to Binary Instruction (cont’d)

The second operand, an eight byte packed decimal number, is converted to fixed-point format and stored in GPR specified by the first operand.

For example: CVB R3, PK1

The 8 byte packed value in PK1 is converted to a fixed point fullword and placed in R3.

Convert to Binary (CVB):
RX Format - Register to indexed storage



Twos complement form of negative number
CC Settings: Condition Code is Unchanged

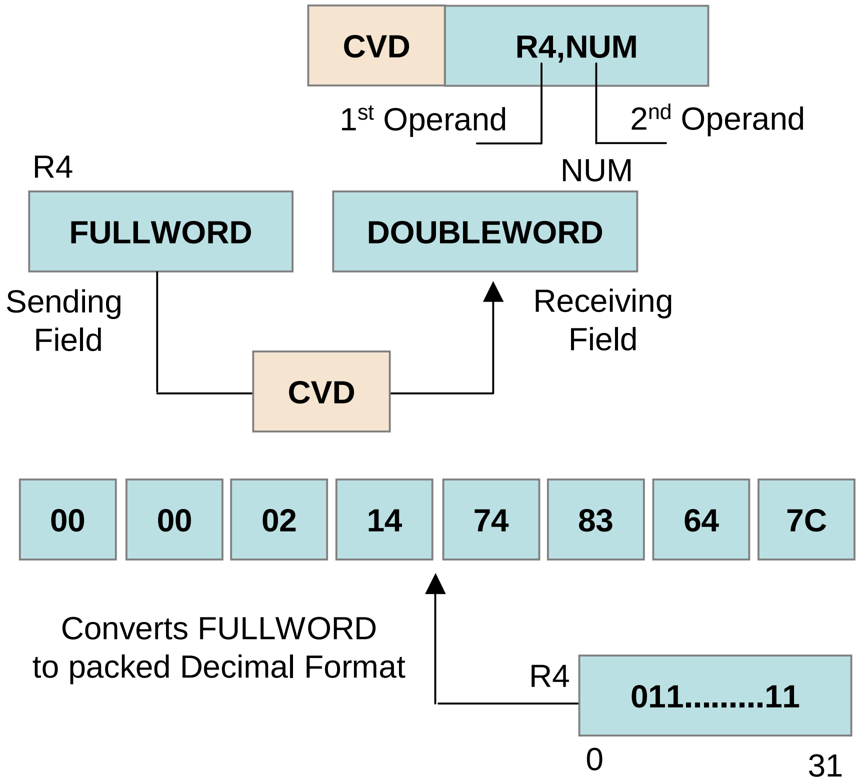
Convert to Decimal Instruction

What is Convert to Decimal instruction?

The Convert to Decimal (CVD) instruction performs the conversion from fixed point to packed format. This is another instruction that works backwards. That is, the second operand is the result field.

The first operand, a fixed point fullword in a register, is converted to an eight byte packed decimal number, at the main storage location specified by the second operand. System /370 and earlier architectures required that the storage location be aligned on a doubleword boundary.

Convert to Decimal (CVD):
RX Format - Register to indexed storage



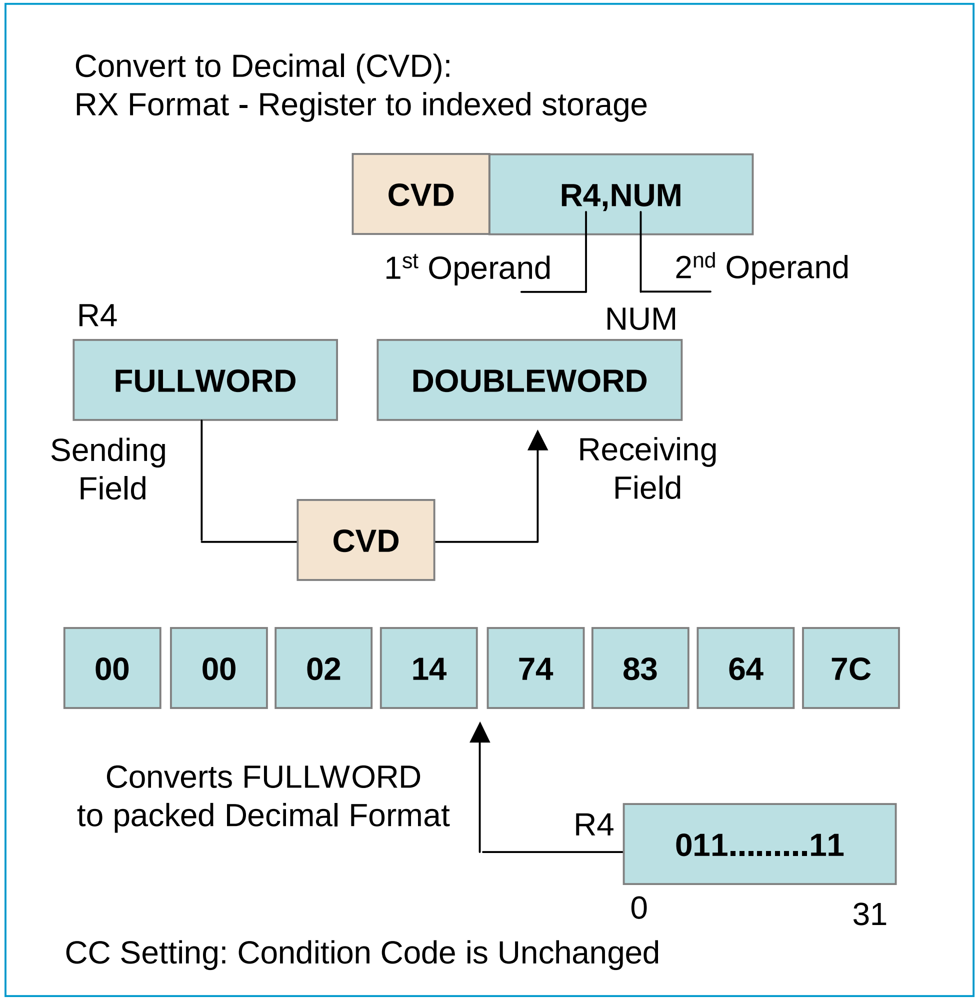
CC Setting: Condition Code is Unchanged

Continued…

Convert to Decimal Instruction (cont'd)

For example: CVD R4,NUM

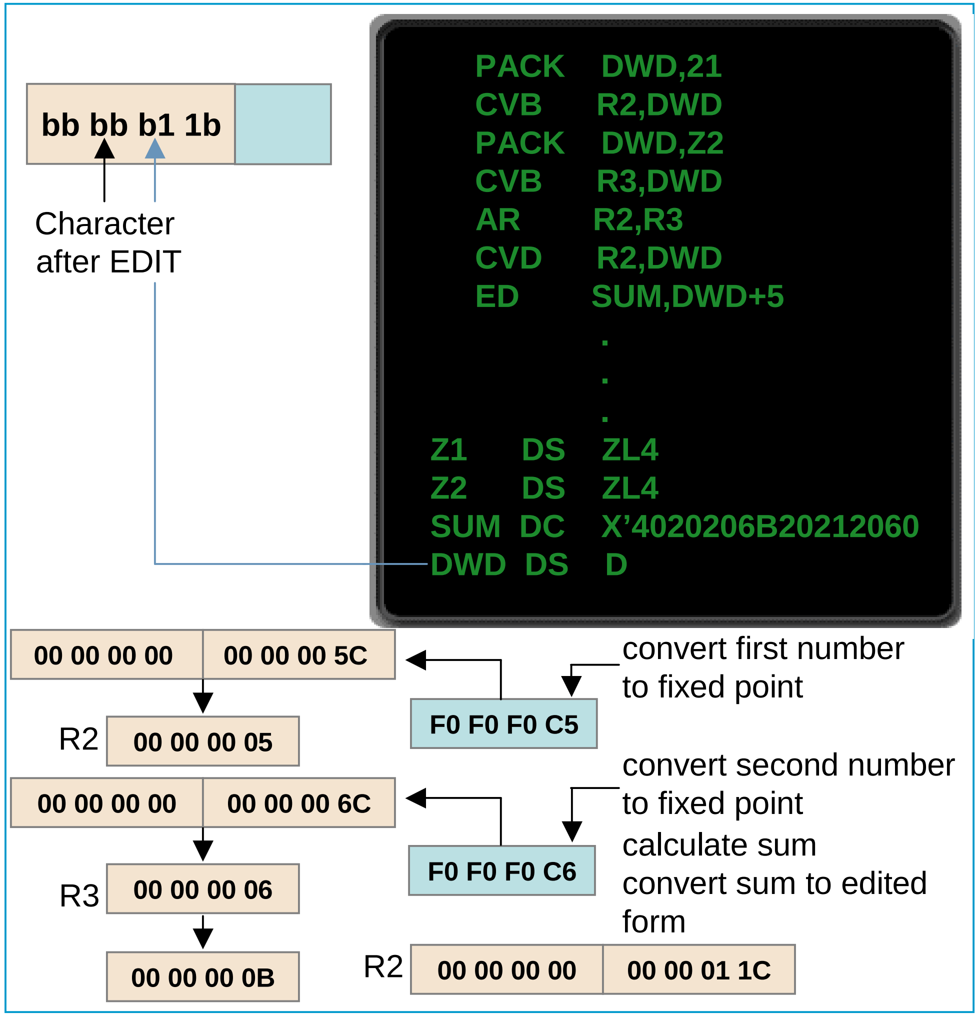
The fullword in R4 is converted to a packed decimal number and stored in main storage in eight bytes beginning at NUM.


Convert to Decimal (CVD):
RX Format - Register to indexed storage
CVD
R4,NUM
1st Operand
2nd Operand
R4
NUM
FULLWORD
DOUBLEWORD
Sending Field
Receiving Field
CVD
00
00
02
14
74
83
64
7C
Converts FULLWORD to packed Decimal Format
R4
011.........11
0
31
CC Setting: Condition Code is Unchanged

Continued…

Convert to Decimal Instruction (cont’d)

The code to take two 4 byte zoned numbers called Z1 and Z2, then calculate their sum using fixed point arithmetic, and then put the sum into an edited field called SUM. The code for this is shown at the right.



```
         PACK  DWD,21
         CVB   R2,DWD
         PACK  DWD,Z2
         CVB   R3,DWD
         AR    R2,R3
         CVD   R2,DWD
         ED    SUM,DWD+5
               .
               .
               .
Z1       DS    ZL4
Z2       DS    ZL4
SUM      DC    X’4020206B20212060’
DWD      DS    D
```

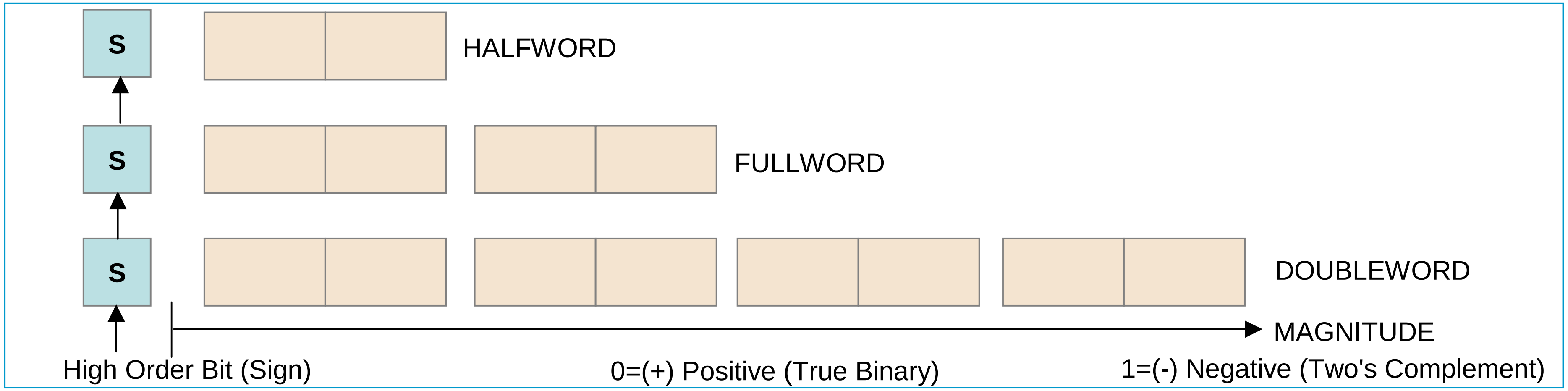
convert first number to fixed point

convert second number to fixed point

calculate sum

convert sum to edited form

Number Representation


S
S
S
HALFWORD
FULLWORD
DOUBLEWORD
MAGNITUDE
High Order Bit (Sign)
0=(+) Positive (True Binary)
1=(-) Negative (Two's Complement)

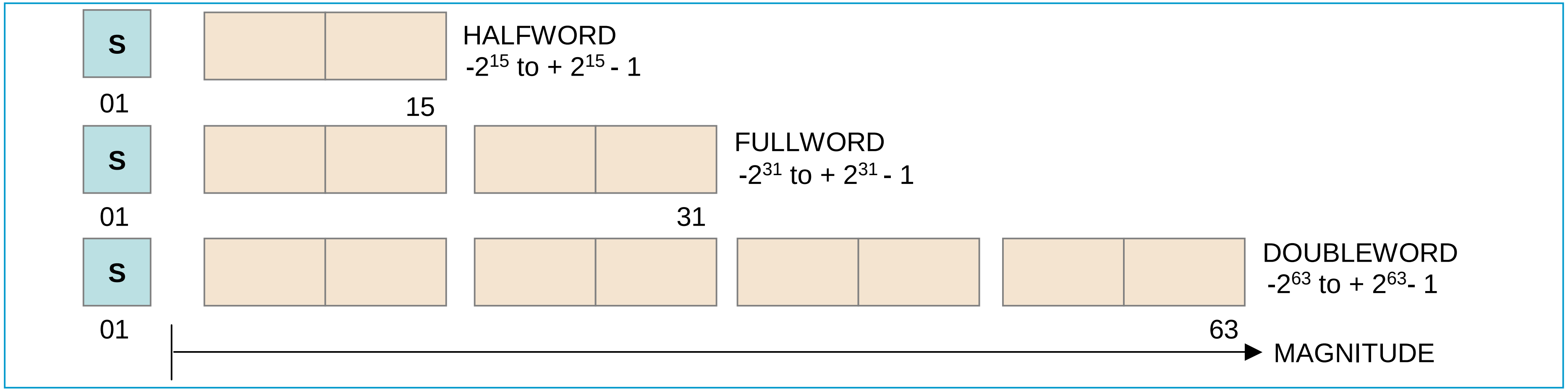
The System/390 architecture provides 3 sets of arithmetic instructions:

- Fixed-point arithmetic instructions deals with integers
- Decimal arithmetic instructions deals with integers
- Floating point arithmetic deals with real numbers (numbers with fractional parts)

Fixed-point arithmetic is performed in the General Purpose Registers (GPRs). Operands for fixed-point arithmetic may be halfwords (16 bits), fullwords (32 bits) or doublewords (64 bits). For all three operand types, the leftmost, or high-order bit, is the sign, with zero representing positive and 1 representing negative. The remainder of the fixed-point number represents the magnitude. Positive numbers are represented in true binary, negative numbers in two's complement form.

Continued…

Number Representation (cont'd)


S
01
15
HALFWORD
-2^{15} to $+ 2^{15} - 1$
S
01
31
FULLWORD
-2^{31} to $+ 2^{31} - 1$
S
01
63
DOUBLEWORD
-2^{63} to $+ 2^{63} - 1$
MAGNITUDE

A halfword has 15 bits to represent the magnitude, and thus can represent values from -2^{15} to $+2^{15}-1$ (32,768 to 32,767).

A fullword has 31 bits to represent the magnitude, and thus can represent values from -2^{31} to $+2^{31}-1$ (-2,147,483,648 to 2,147,483,647).

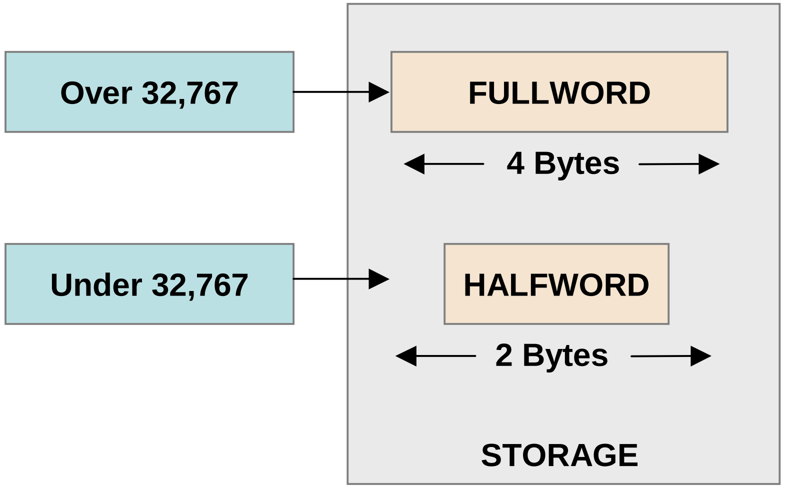
A doubleword has 63 bits to represent the magnitude, and thus can represent values from -2^{63} to $2^{63}-1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807).

Continued…

Number Representation (cont'd)

There is not a single instruction to handle all data types for all arithmetic operations. As you look at the individual instructions, you will see what operand-operation combinations are allowed.

The Assembler Language programmer chooses a particular fixed-point data type for each variable, based on the range of values the variable can take on, and the operations to be performed on it. Generally speaking, using the smallest data type that provides the range of values you require will result in the smallest and fastest program.



HALFWORD
produces a smaller
more efficient program.
It uses 2 bytes rather then
4 bytes in storage.

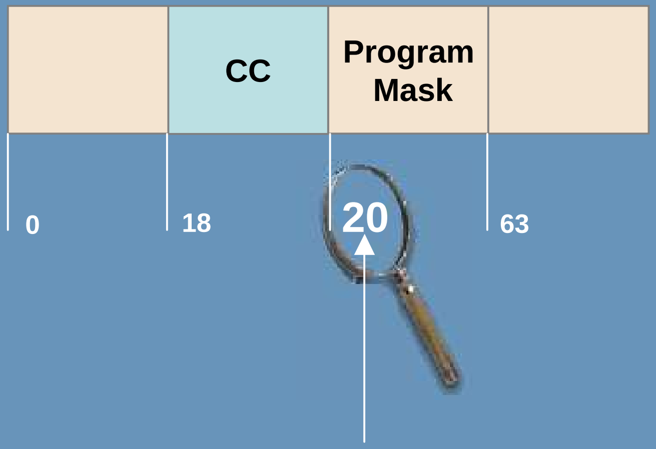
Unit: Fixed-Point Binary Arithmetic Topic: Addition and Subtraction

The Name Field

If you choose a data type that is too small, you may cause a condition known as overflow. Overflow occurs when a generated result is too large to fit into its destination field.

The setting of the fixed-point overflow bit in the program work field of the PSW (Bit no. 20) indicates that a program interrupt will occur causing an abnormal termination. If this bit is not set, the program may seem to execute correctly but produce incorrect numeric results.

Program Status Word (PSW) EC Mode:



Fixed-point Overflow Bit

On - Abnormal Termination
Off - Sets Condition Code
and carries on

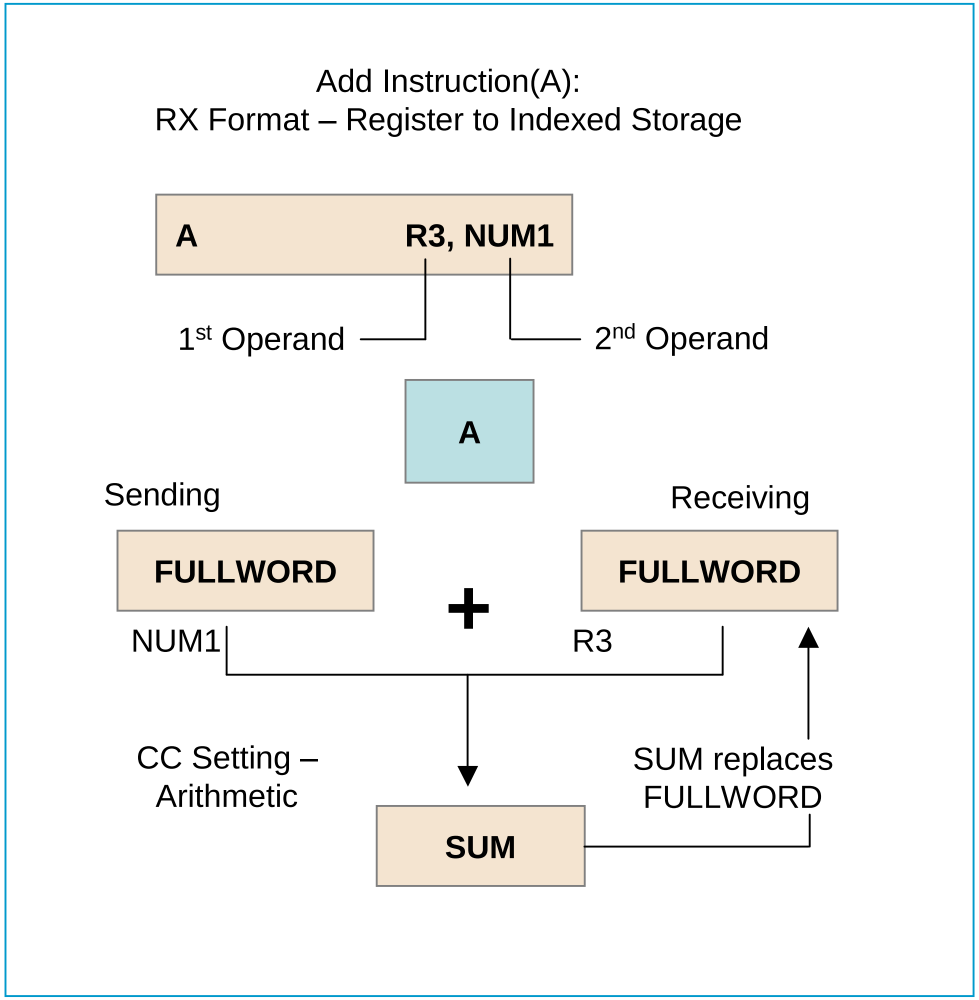
Add Instruction

The first group of fixed-point arithmetic instructions to consider are the addition instructions. There are three instructions in this group: A, AH and AR.

The Add Instruction (A) adds a fullword in main storage to a fullword in a GPR, with the sum replacing the value in the GPR.

For example: A R3,NUM1

In this case, the fullword at NUM1 would be added to the fullword in R3, and the sum would be placed in R3.



Continued…

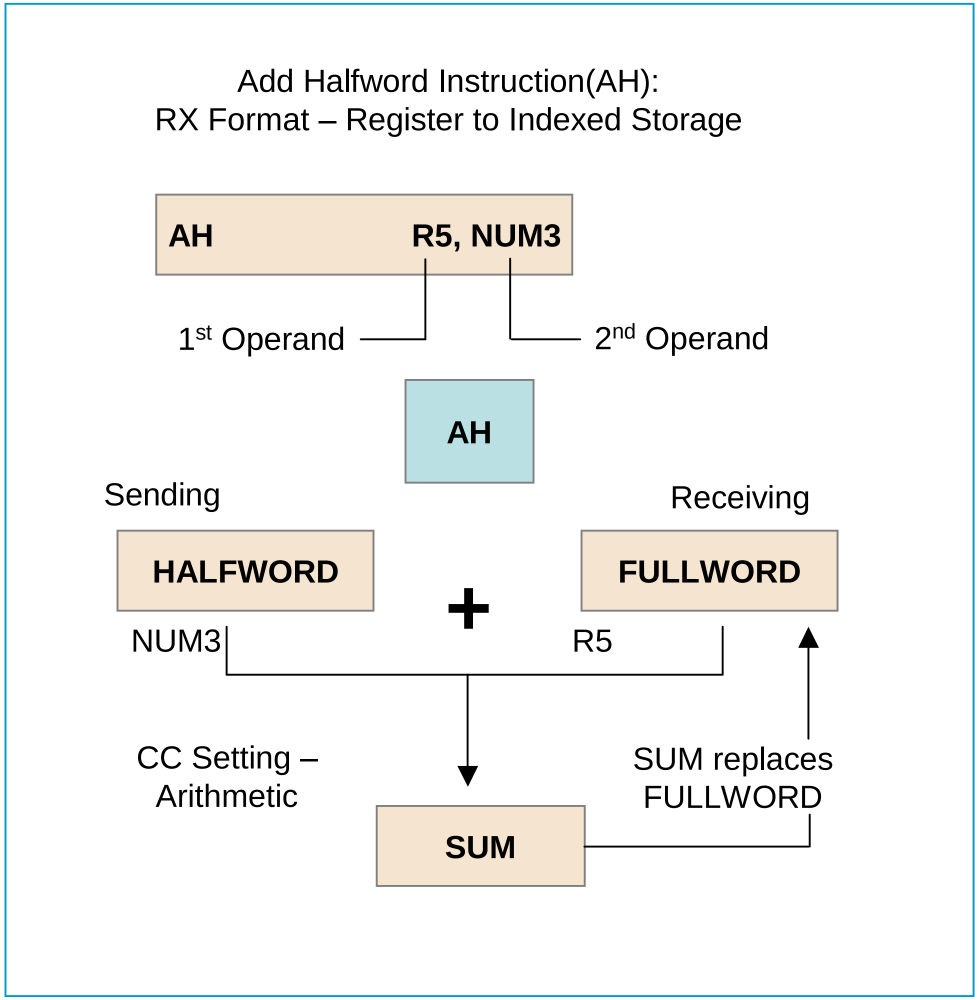
Unit: Fixed-Point Binary Arithmetic Topic: Addition and Subtraction

Add Halfword Instruction

The Add Halfword Instruction (AH) is similar to Add (A), except that the second operand is a halfword in main storage. The first operand is the whole 32 bits in the register, and the sum is a fullword.

For example : AH R5,NUM3

In executing this instruction, the halfword at NUM3 would be added to the fullword in R5, and the sum placed in R5.



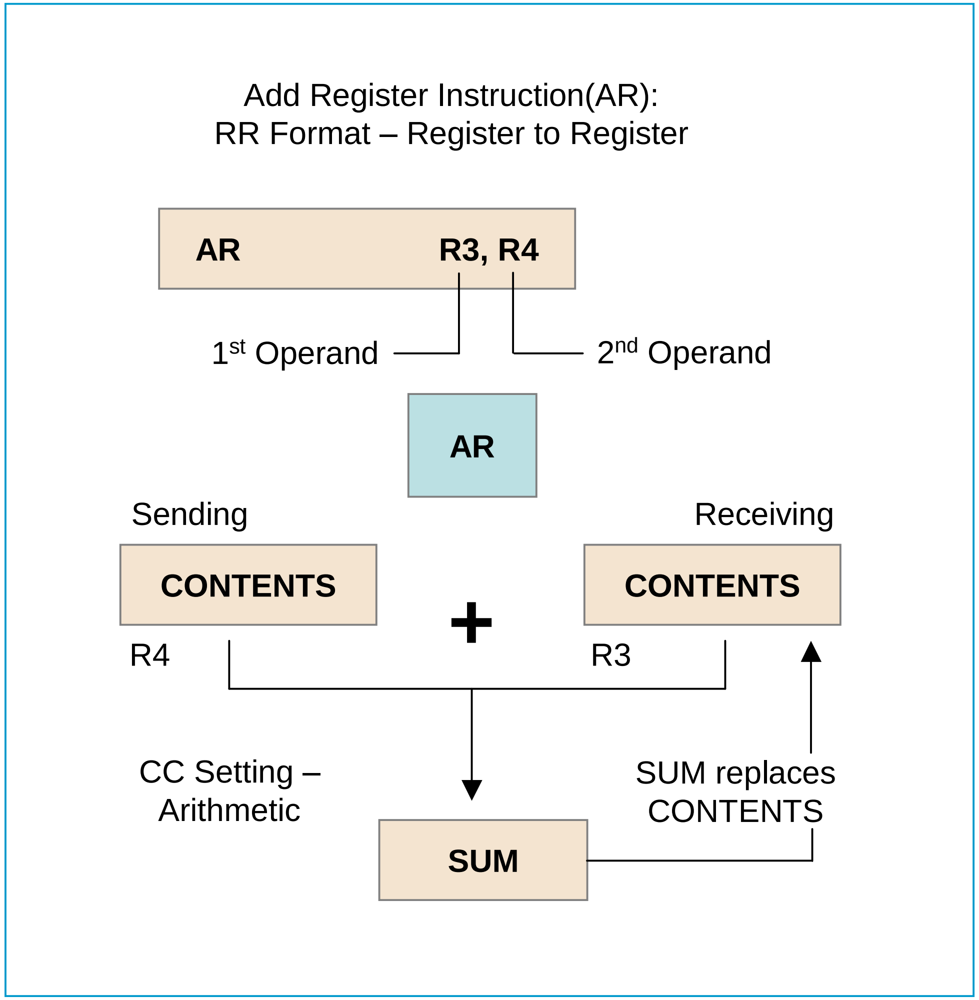
Continued…

The Add Register Instruction

The fullword in the second operand register is added to the fullword in the first operand register, and the sum is placed in the first operand register. The Add Register (AR) Instruction, is similar to Add (A), except that the second operand is in a register instead of main storage.

For example: AR R3,R4

The contents of R4 are added to the contents of R3, with the sum being placed in R3.



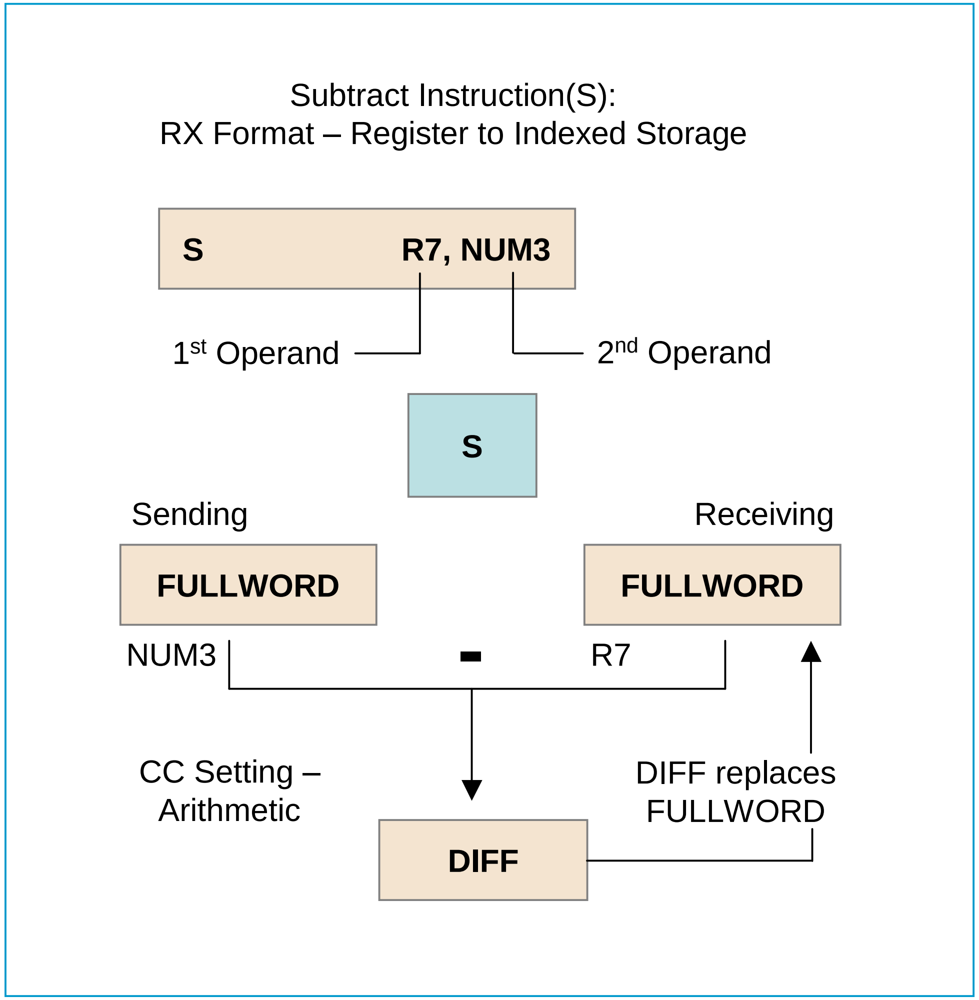
Subtract Instruction

There are also three fixed-point subtraction instructions: S, SH and SR. They operate in a similar way to the Add instructions, except that the operation is subtraction rather than addition.

The Subtract Instruction (S) subtracts the second operand, a fullword in a main storage, from the first operand, a fullword in a GPR, and the difference is placed in the register.

For example: S R7,NUM3

In the example on the right, the fullword at NUM3 is subtracted from the fullword in R7, and the difference is placed in R7.

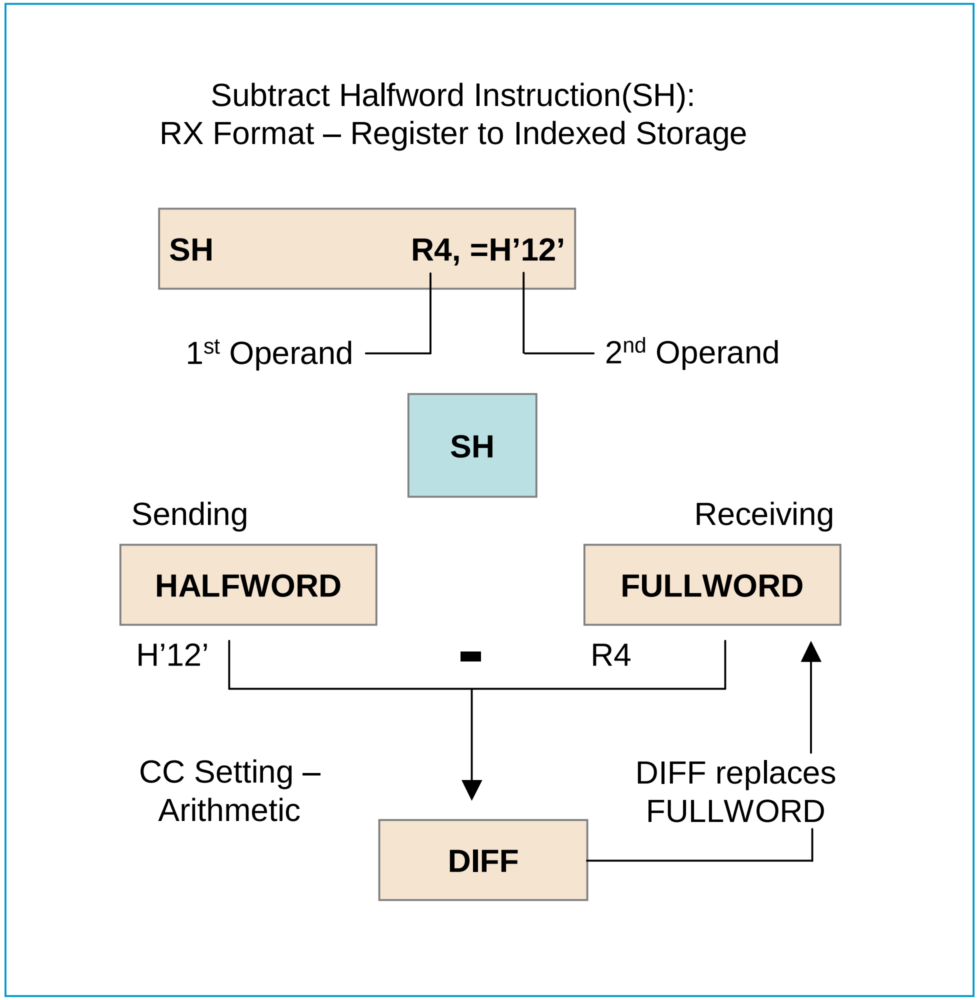


Subtract Halfword Instruction

The Subtract Halfword Instruction (SH) subtracts the halfword at the main storage location specified by the second operand, from the fullword in the GPR specified by the first operand. The difference is placed in the GPR.

For example: SH R4,=H'12'

In the example on the right, the SH instruction subtracts 12 (specified as a literal) from the value in R4, and the difference is placed in R4.

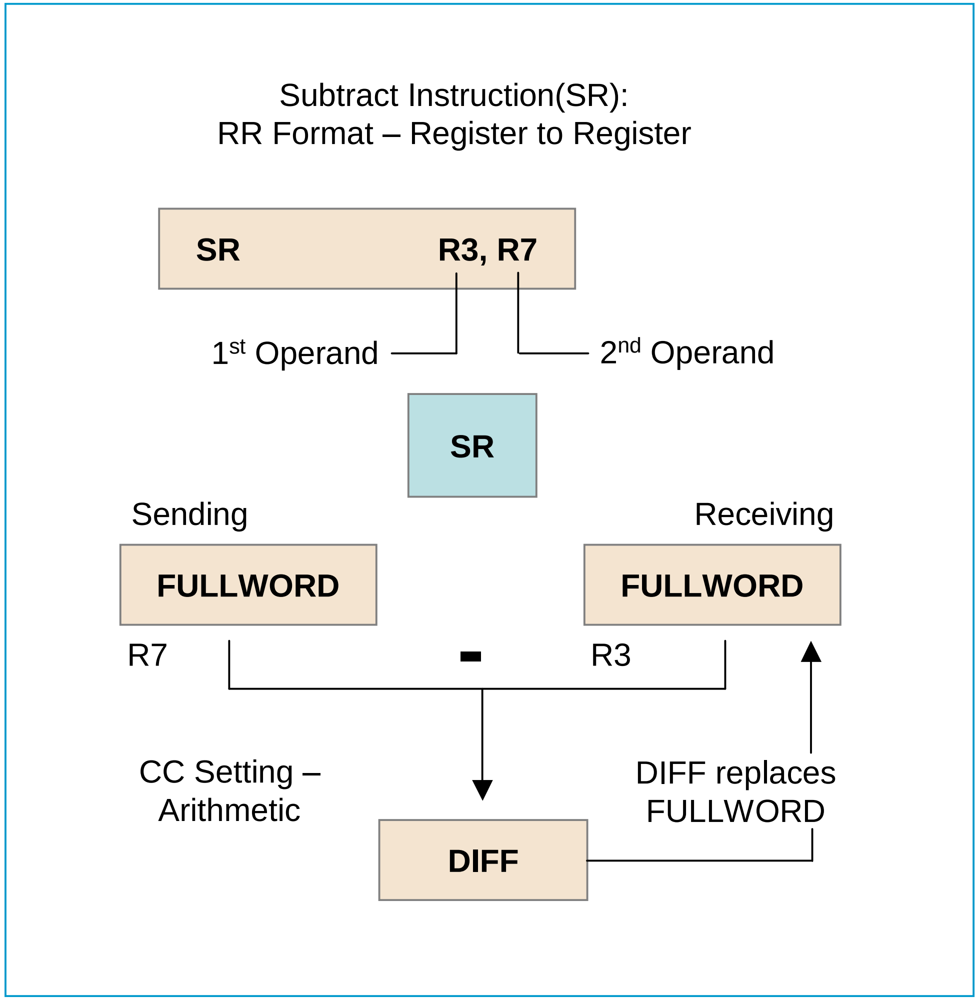


Subtract Register Instruction

The Subtract Instruction (SR) subtracts a fullword in the second operand register, from a fullword in the first operand register, and the difference is placed in the first operand register.

For example: SR R3,R7

In the example on the right, the fullword in R7 is subtracted from the fullword in R3, with the difference placed in R3.



Continued…

Subtract Register Instruction (cont'd)

One application of SR is to zero a register. You could set a register to zero by loading a fullword of zero into it.

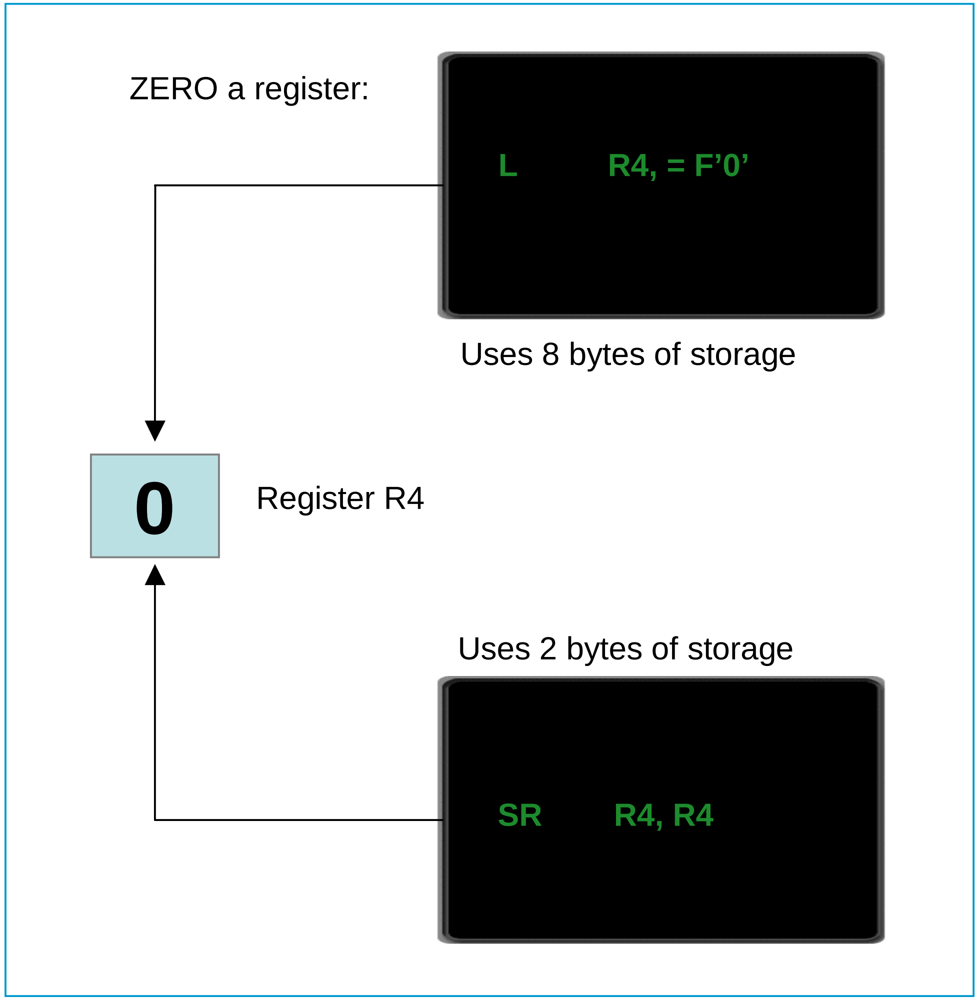
For example: L R4,=F'0'

This method requires 4 bytes for the instruction (L is an RX instruction) plus 4 bytes for the literal.

A better way to set R4 to zero is by subtracting its contents from itself.

```
     SR    R4,R4
```

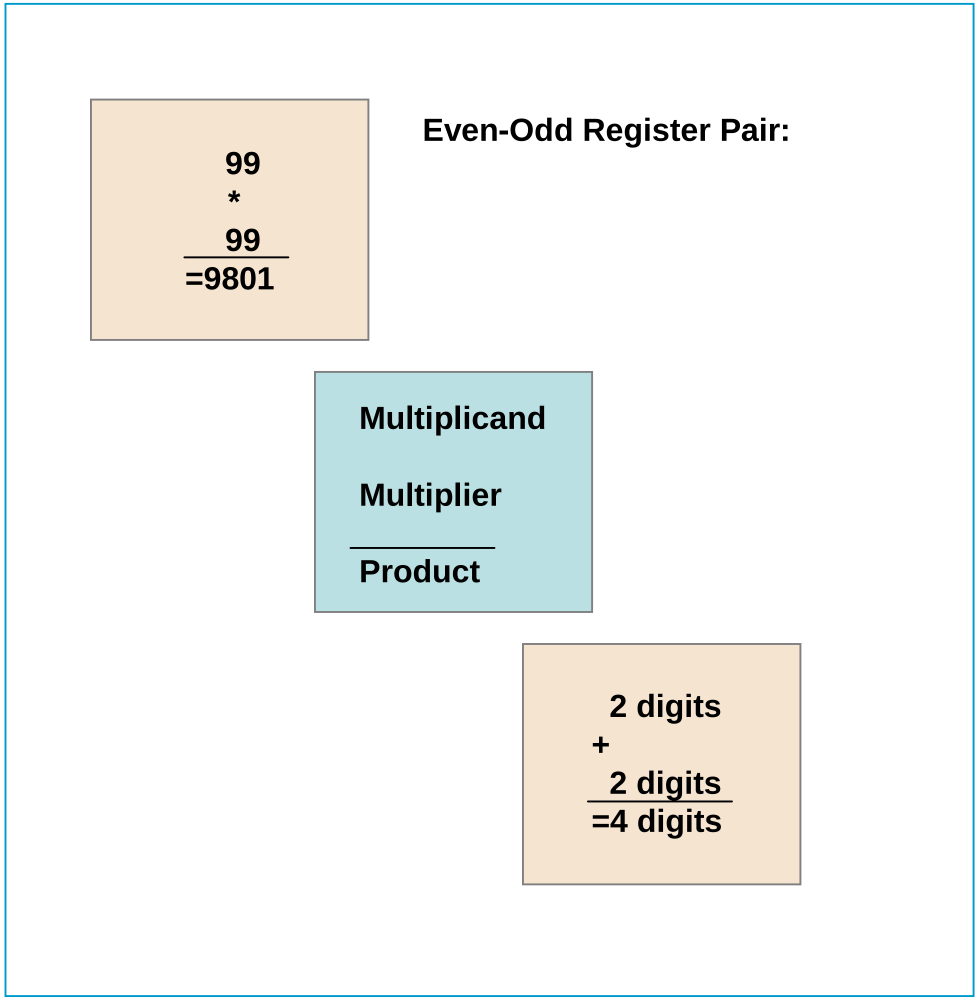
Regardless of the initial value of R4, the result will be zero. This method only uses 2 bytes of storage, for the RR instruction.



Multiplication

When you perform addition and subtraction, the result is of roughly the same magnitude, or less than the largest of the operands. Thus, it makes sense that when you are adding two fullwords, the sum also be a fullword. However, when you do multiplication, the number of digits in the product is roughly the sum of the number of digits in the multiplicand and multiplier.

The multiplier and multiplicand have two digits each, and the products has four digits. It makes sense, then, that when a multiply instruction is designed as a part of a computer architecture, it should provide for a product larger than the multiplier and the multiplicand.



Continued…

Multiplication (cont’d)

There are three fixed-point multiplication instructions. They are:

- MH is analogous in operation to AH and SH
- M and MR use the first operand GPR specification in a non-intuitive way that requires some explanation

Even-Odd Register Pair:

```
   99
×  99
=9801
```

2 digits	**Multiplicand**
+2 digits	**Multiplier**
= 4 digits	**Product**

The Product is larger then both the Multiplicand and the Multiplier. The hardware requires that the Product forms an EVEN-ODD Register Pair.

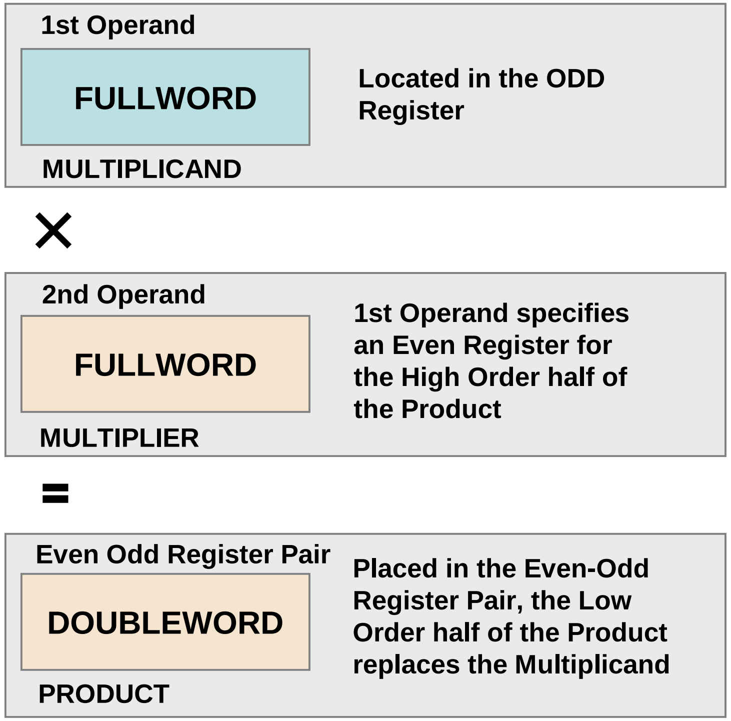
Continued…

Unit: Fixed-Point Binary Arithmetic Topic: Multiplication

Multiplication (cont’d)

For both M and MR, the multiplicand and multiplier are fullwords. The product is a doubleword. The first operand must specify an even numbered GPR. The multiplicand must be located in the odd numbered GPR, one greater than the first operand specification.

The multiplier is the second operand. The product (64 bits) is placed in the even-odd register pair, wiping out the multiplicand in the odd register of the pair.



1st Operand

FULLWORD

MULTIPLICAND

Located in the ODD Register

×

2nd Operand

FULLWORD

MULTIPLIER

1st Operand specifies an Even Register for the High Order half of the Product

=

Even Odd Register Pair

DOUBLEWORD

PRODUCT

Placed in the Even-Odd Register Pair, the Low Order half of the Product replaces the Multiplicand

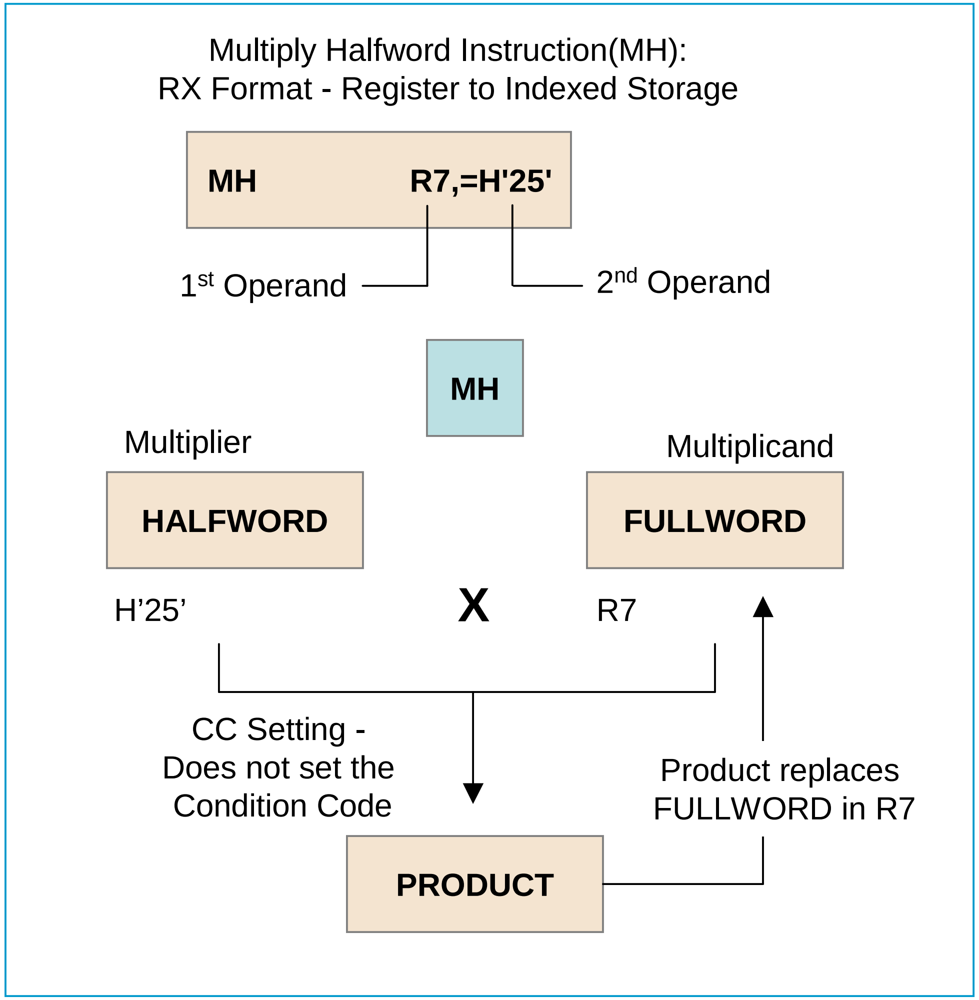
Multiply Halfword Instruction

What is the Multiply Halfword Instruction?

The Multiply Halfword Instruction (MH) multiplies a halfword in main storage (which is the Multiplier specified by the second operand), by the 32 bit multiplicand in the register specified as the first operand. The low order 32 bits of the product replace the multiplicand. If overflow occurs, significant bits to the left of the 32 saved bits, are lost, but there is no indication of this truncation.

For example: MH R7,=H'25'

The value in R7 is multiplied by 25, and the product placed in R7.



Note!

Multiply and divide instructions do not set the condition code.

Multiply Instruction

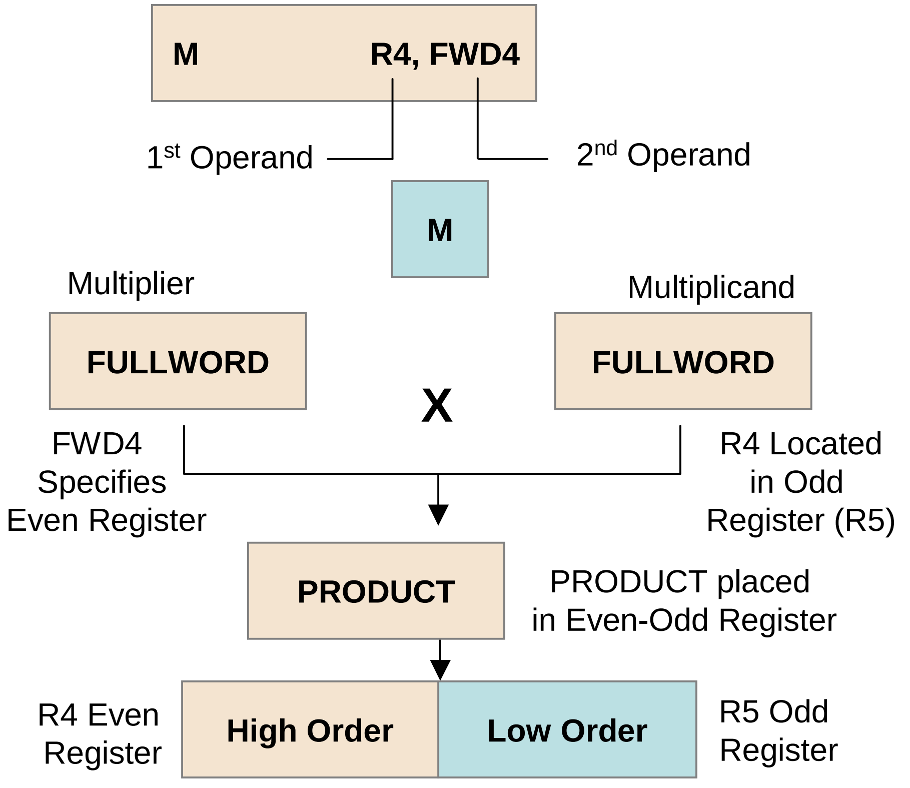
What is the Multiply Instruction?

The Multiply Instruction (M) multiplies a fullword in the GPR (one greater than the first operand register specified), by a fullword second operand storage. The product, a 64 bit fixed point number is placed in the first operand register, and in the register one greater. The first operand must specify an even numbered register, or a specification exception will occur. It is impossible for overflow to occur.

For example: M R4, FWD4

The fullword in R5 is multiplied by the fullword at main storage location FWD4. The 64-bit product is placed in GPRs 4 and 5.

Multiply Instruction (M):
RX Format - Register to Indexed Storage



CC Setting - Does not set the Condition Code

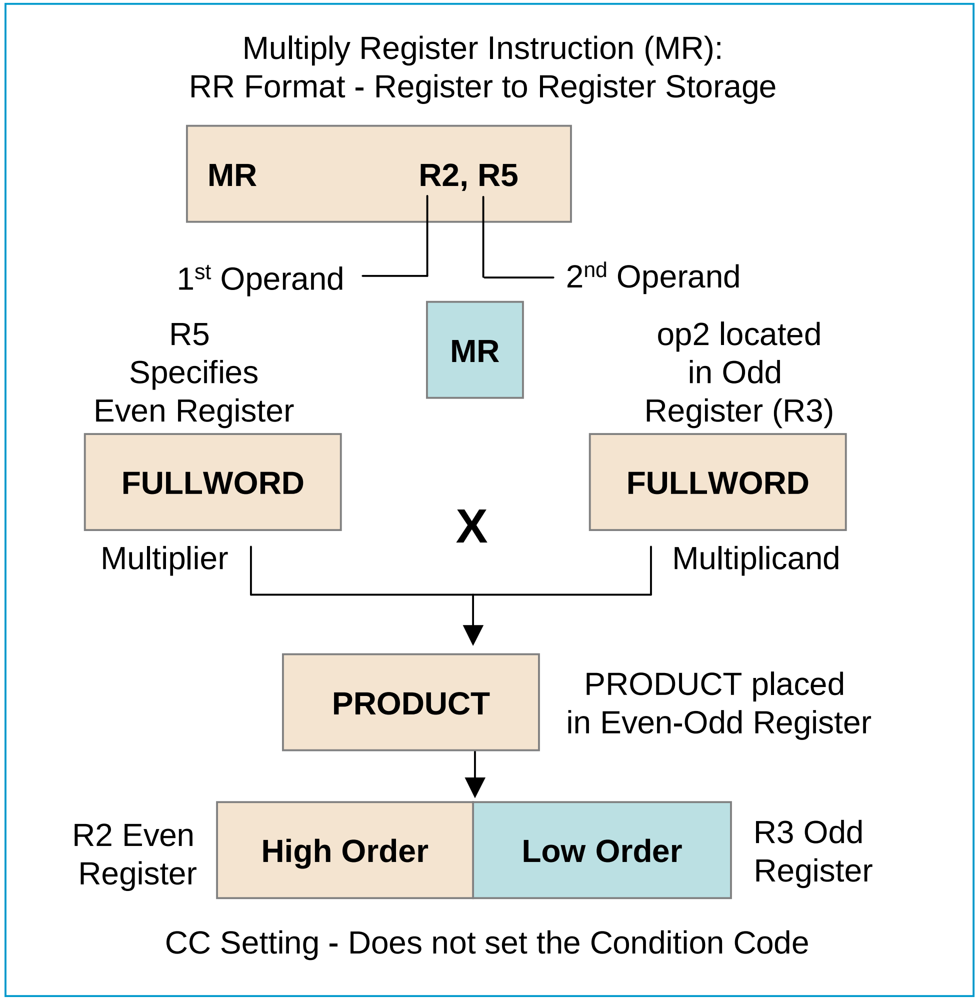
Multiply Register Instruction

What is the Multiply Register Instruction?

The Multiply Register Instruction (MR) multiplies a fullword in the GPR one greater than the first operand register specified, by the fullword in the register specified as the second operand. The product, a 64 bit fixed point number, is placed in the first operand register, and in the register one greater. The first operand must specify an even numbered register.

For example: MR R2,R5

The fullword in R3 is multiplied by the fullword in R5. The 64 bit product is placed in GPRs 2 and 3.


Multiply Register Instruction (MR):
RR Format - Register to Register Storage
MR R2, R5
1st Operand
2nd Operand
R5 Specifies Even Register
MR
op2 located in Odd Register (R3)
FULLWORD
X
FULLWORD
Multiplier
Multiplicand
PRODUCT
PRODUCT placed in Even-Odd Register
R2 Even Register
High Order
Low Order
R3 Odd Register
CC Setting - Does not set the Condition Code

Continued…

Unit: Fixed-Point Binary Arithmetic Topic: Division

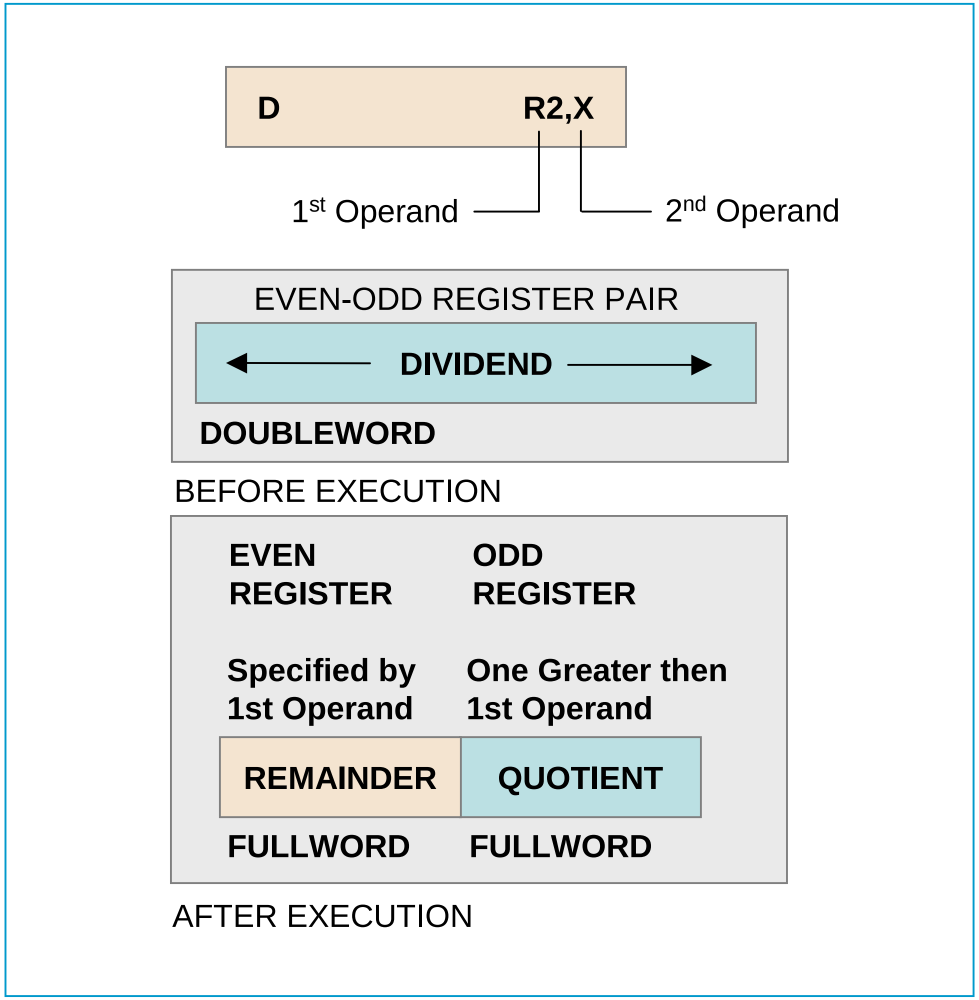
Fixed-Point Division

What are the two fixed-point division operations?

There are only two fixed-point division operations, D and DR. There is no DH instruction.

Fixed-point arithmetic is integer arithmetic, so division produces both an integer quotient and an integer remainder. The dividend is a 64-bit quantity in an even-odd register pair. The divisor is a fullword.

After the division, the remainder is placed in the even numbered register, specified by the first operand, and the quotient goes into the odd numbered register one greater than the first operand.

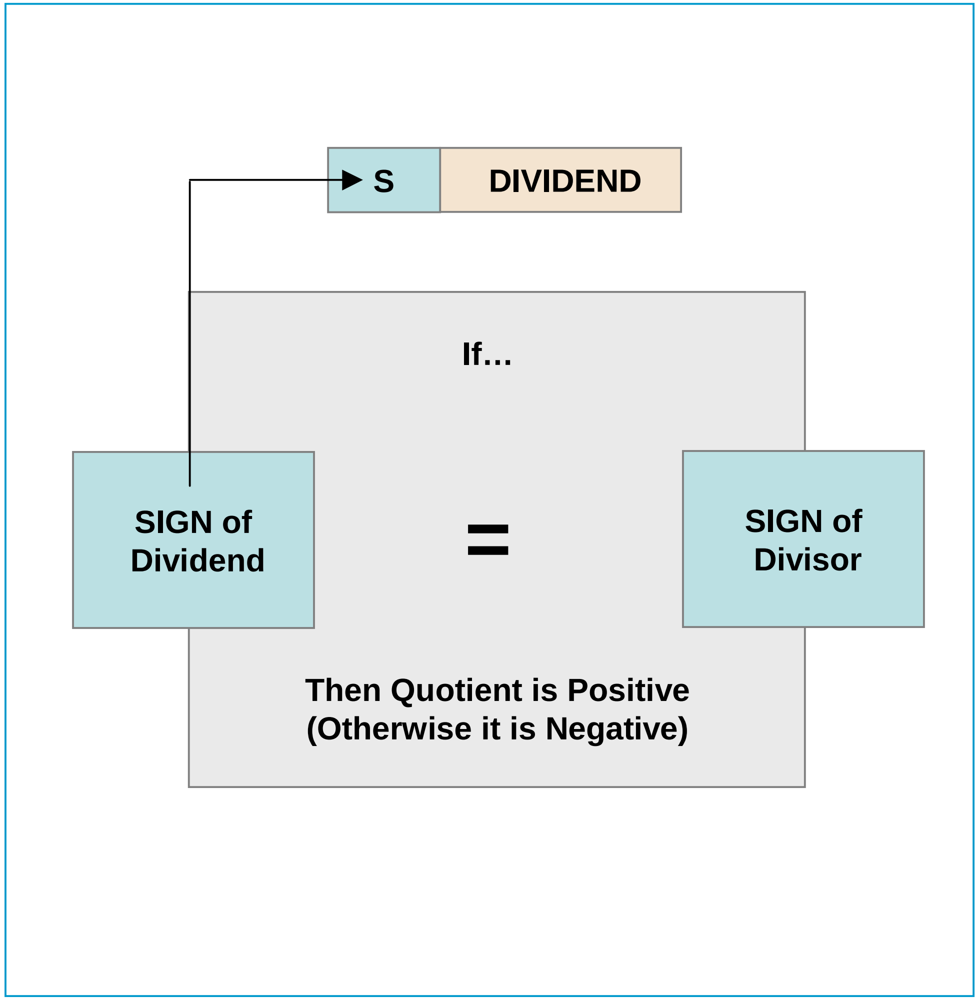


Continued…

Fixed-Point Division (cont'd)

The sign of the quotient is determined by the rules of algebra. That is, if the signs of dividend and divisor are the same, the quotient is positive, otherwise it is negative. The sign of the remainder is the same as the sign of the dividend.

If the divisor is zero, or the quotient is too large to fit into a fullword, a fixed-point divide exception occurs. This will cause your program to terminate abnormally.



Signs of Quotient and Remainder

DIVIDEND	DIVISOR	QUOTIENT	REMAINDER
17	5	3	2
-17	5	-3	-2
17	-5	-3	2
-17	-5	3	-2
15	5	3	0

The table above indicates the results of some division operations, showing which signs are generated for both quotient and remainder.

Divide Instruction

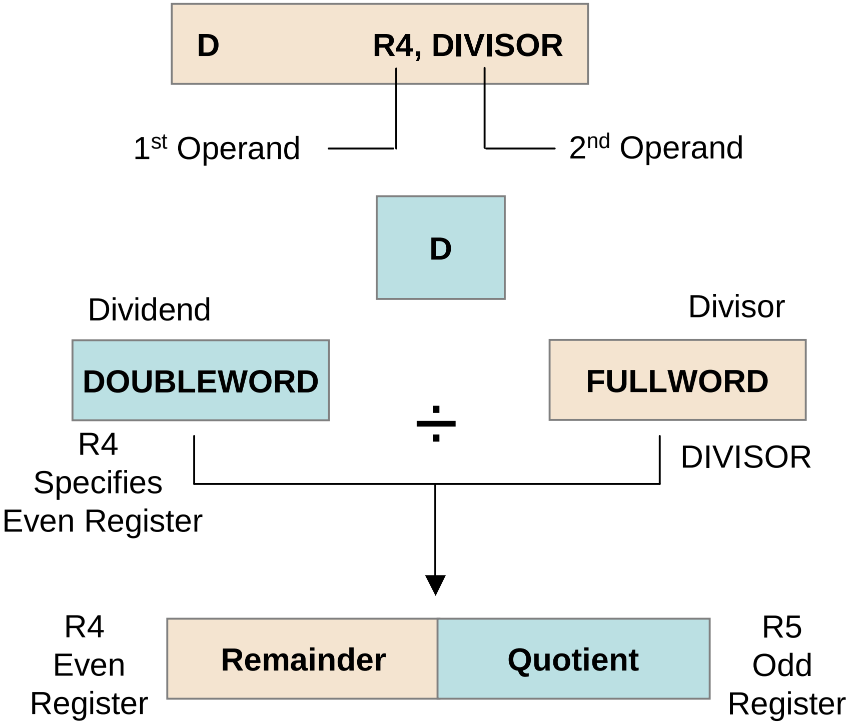
What is the Divide instruction?

The Divide Instruction (D) divides the 64 bit fixed point number in the even-odd register pair specified by the first operand (which must be even), by the fixed point fullword in the storage location specified by the second operand. The result consists of a fullword remainder that is placed in the even numbered register (R_1), and a fullword quotient which is placed in the odd numbered register (R_1 +1).

For example: D R4, DIVISOR

The 64 bit fixed point number in registers R4 and R5 is divided by the fullword at DIVISOR, and the integer remainder is placed in R4 and the integer quotient in R5.

Divide Instruction(D):
RX Format – Register to Indexed Storage

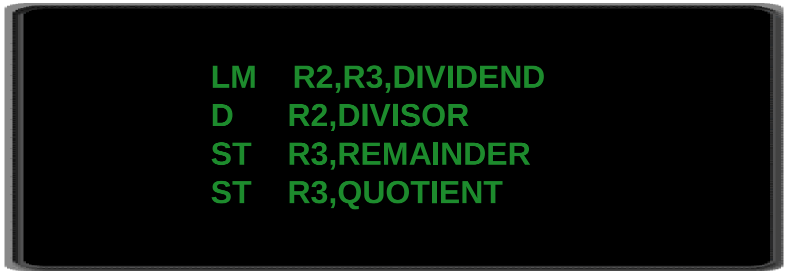


CC Setting – Does not set the Condition Code

Continued…

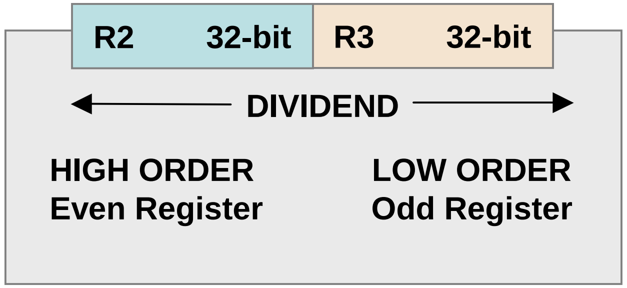
Divide Instruction (cont’d)

If you wanted to divide a 64-bit variable called DIVIDEND, by a fullword variable called DIVISOR, to produce fullword QUOTIENT and REMAINDER, you could use the code shown on the right.



```
LM    R2,R3,DIVIDEND
D     R2,DIVISOR
ST    R3,REMAINDER
ST    R3,QUOTIENT
```

LOAD MULTIPLE



DIVIDE - (R2,R3) Dividend by Divisor

STORE - Remainder into Even Register R2

Quotient into Odd Register R3

Unit: Fixed-Point Binary Arithmetic Topic: Division

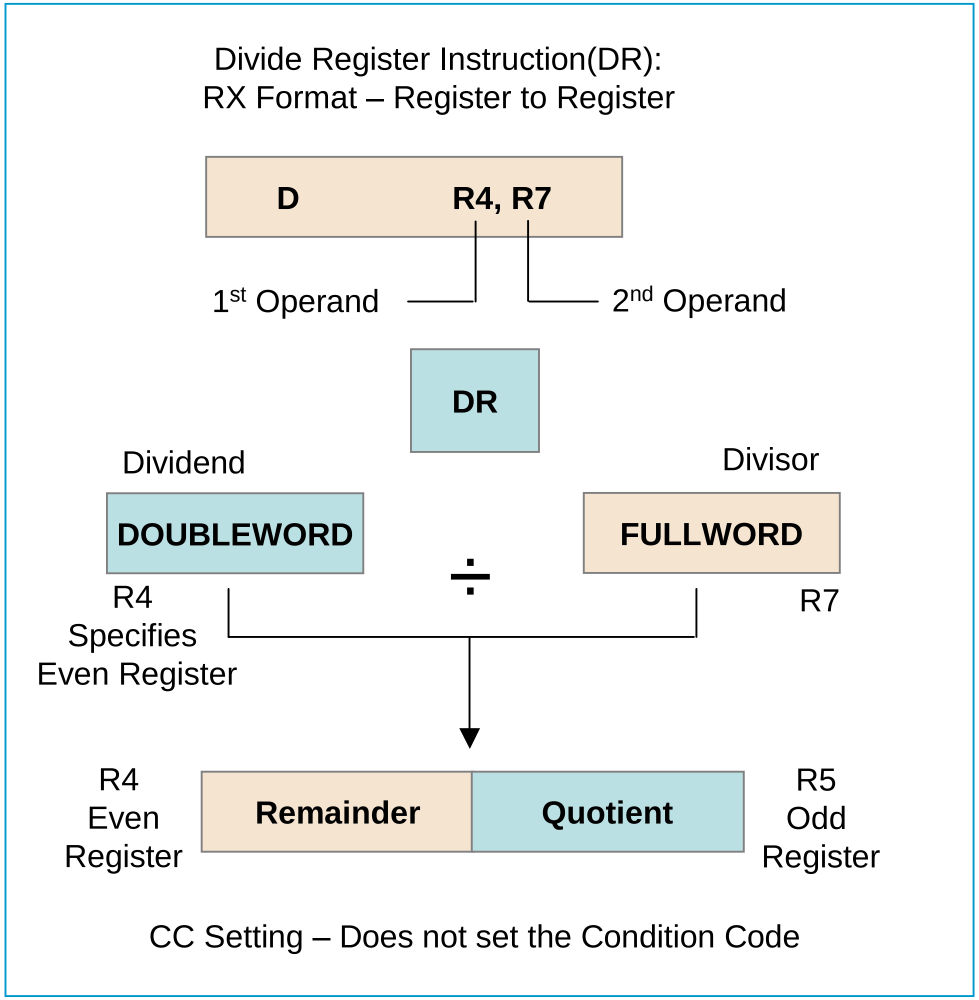
Divide Register Instruction

What is the Divide Register instruction?

The Divide Register Instruction (DR) divides a 64 bit fixed point number in the even-odd register pair specified by the first operand (which must be even), by the fixed point fullword in the GPR specified by the second operand. The result consists of a fullword remainder that is placed in the even numbered register (R_1), and a fullword quotient that is placed in the odd numbered register (R_1+1).

For example: DR R4,R7

The 64 bit fixed point number in registers R4 and R5 is divided by the fullword in R7, and the integer remainder is placed in R4 and in the integer quotient in R5.



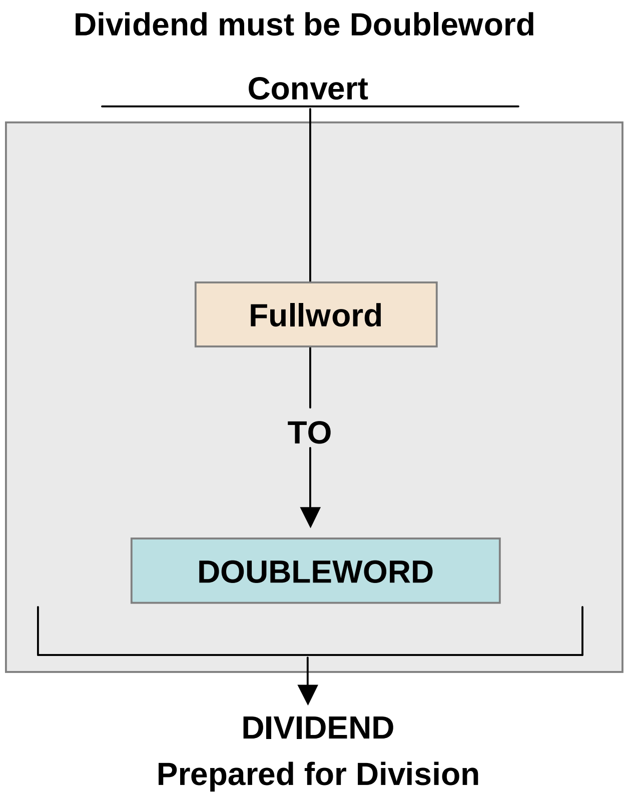
Converting From Fullword to Doubleword for Division

What are the three ways of converting from fullword to doubleword for division?

In performing fixed-point arithmetic, you are often working with fullword operands.

There are many times when you will want to perform division on a value that is in a fullword variable. However, the division instructions require that the dividend be a doubleword.

You must convert the fullword to a doubleword prior to performing the division. There are three standard ways that this fullword to doubleword conversion can be done.

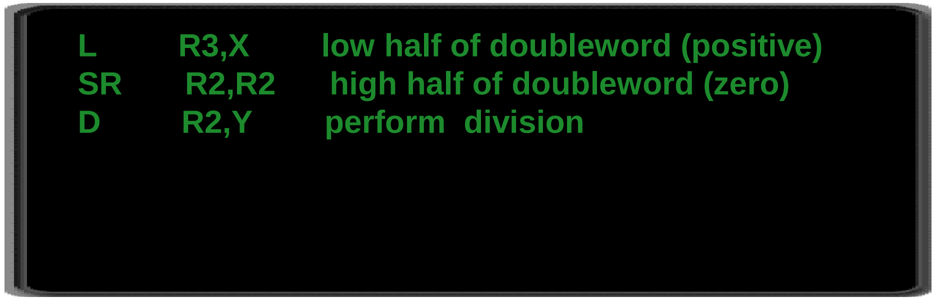


Continued…

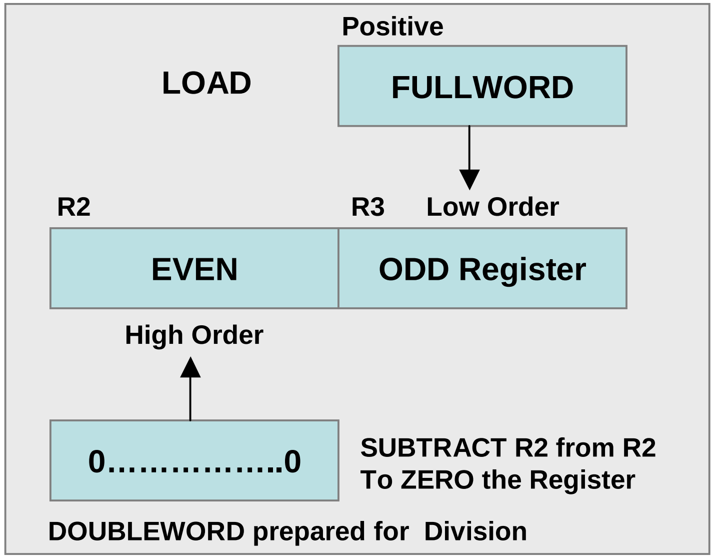
Converting From Fullword to Doubleword for Division (cont’d)

First Method: Load (L) / Subtract Register (SR)

For the first method, you are only working with positive numbers, and the first half of the doubleword will be binary zeros. Therefore, just load the fullword into the odd register and set the even register to zero. To divide fullword X by fullword Y, you could code as shown on the right.



```
L       R3,X      low half of doubleword (positive)
SR      R2,R2     high half of doubleword (zero)
D       R2,Y      perform  division
```



Continued…

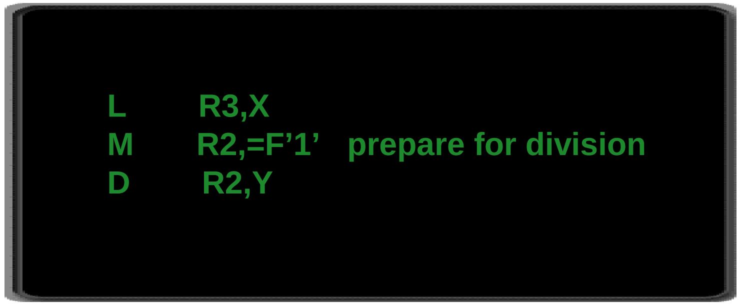
Converting From Fullword to Doubleword for Division (cont'd)

Second Method: Multiplication

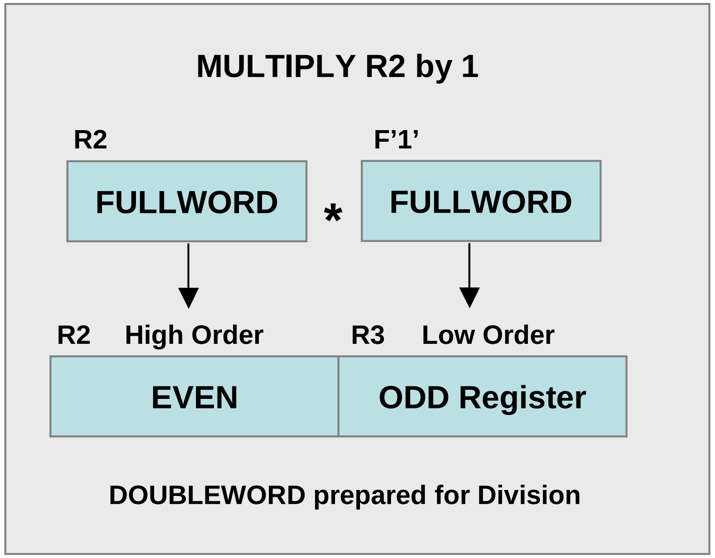
The second and third methods will handle both positive and negative dividends.

The second method uses multiplication. Remember that multiplication multiplies two fullwords, giving a 64-bit result.

If you multiply a fullword value by one, you do not change its value, but u do change it to a 64-bit quantity. Using this method to divide X by Y, you could use the code shown on the right.



```
L       R3,X
M       R2,=F'1'   prepare for division
D       R2,Y
```

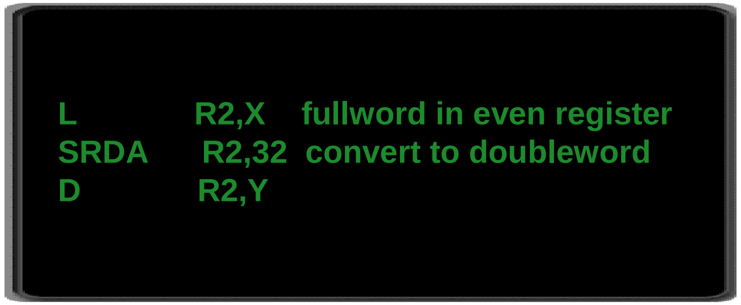


Continued…

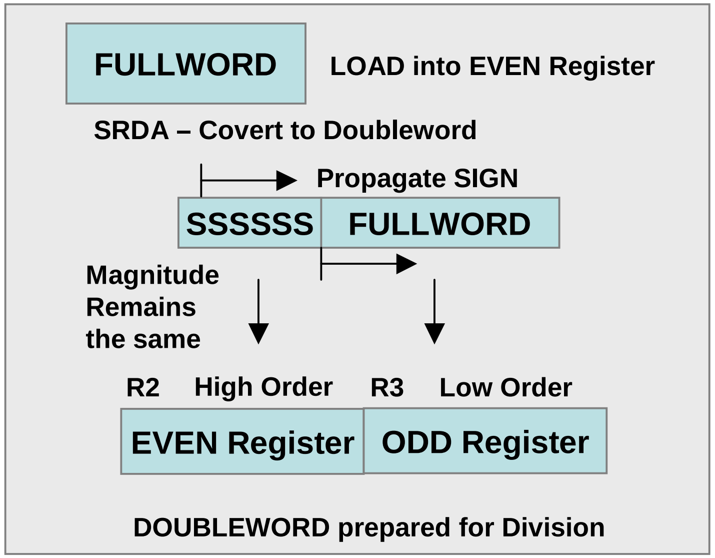
Converting From Fullword to Doubleword for Division (cont'd)

Third Method: Shift Right Double (SRDA)

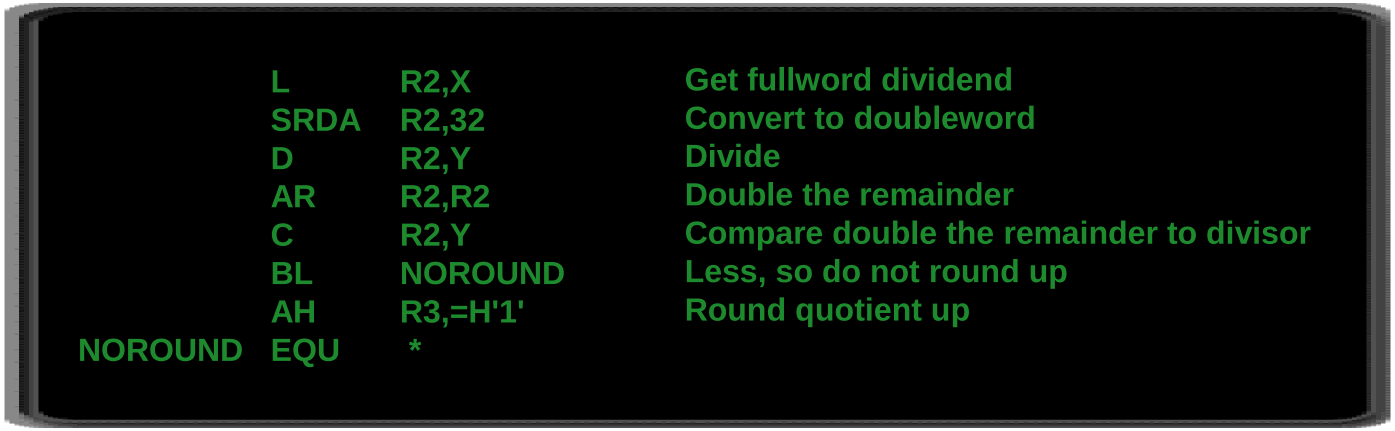
The third method uses an instruction you have not seen yet, SRDA (shift right double). Using this instruction, the fullword dividend is loaded into the even numbered register of the even-odd pair, and then shifted (double) 32 bits, producing a 64-bit dividend. With this third method, the code to perform the division is shown on the right.



```
L       R2,X    fullword in even register
SRDA    R2,32   convert to doubleword
D       R2,Y
```



Rounding After Division



```
         L        R2,X          Get fullword dividend
         SRDA     R2,32         Convert to doubleword
         D        R2,Y          Divide
         AR       R2,R2         Double the remainder
         C        R2,Y          Compare double the remainder to divisor
         BL       NOROUND       Less, so do not round up
         AH       R3,=H'1'      Round quotient up
NOROUND  EQU       *
```

There is one more issue you should consider with regards to division. Often, you decide that you do not want a remainder per se, but rather a rounded quotient. That is, if the remainder is equal to or greater than half of the divisor, you increase the quotient by one. To round, perform the division, double the remainder and compare it to the divisor. If it is equal or greater, you add one to the quotient. The code is shown above.

Uses of Iterative Structure

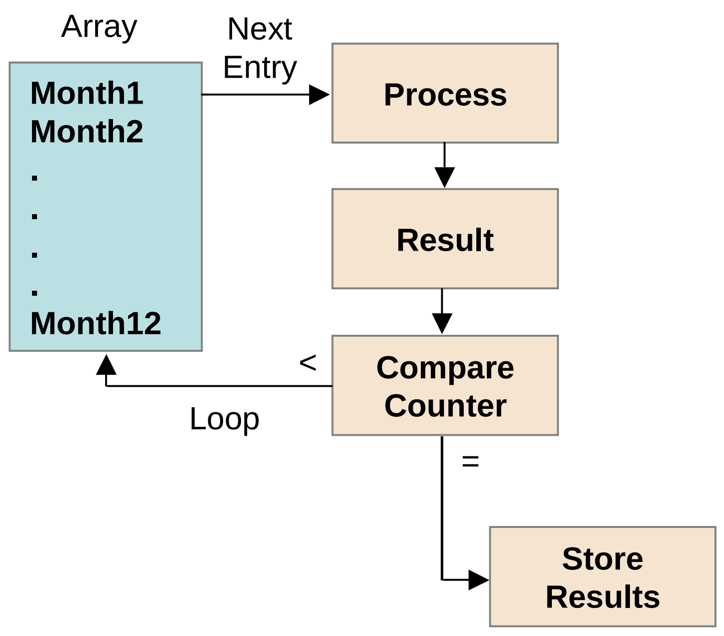
What is looping?

Looping is a coding technique that enables to repeatedly execute the same sequence of machine instructions.

What is an iterative loop?

Unlike situations where the end condition is unknown until execution time, an iterative loop is coded when its known how many times the loop is needed.

The example shows processing of an array with one entry for each month.



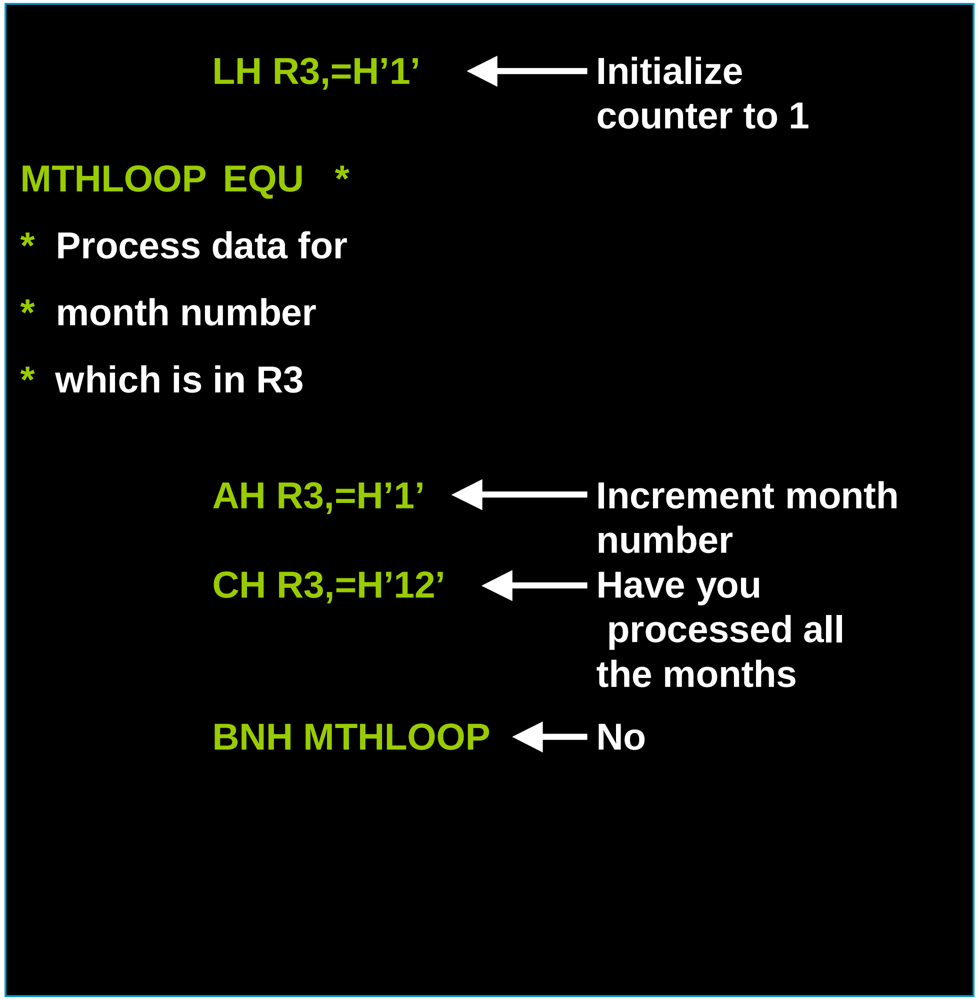
Continued…

Uses of Iterative Structure (cont'd)

How is an iterative loop built?

Iterative loop structure can be built by using a counter and a comparison followed by a Branch on Condition.

The example shows the code used to process the array of 12 months.



```
            LH R3,=H'1'     ←  Initialize
                               counter to 1
MTHLOOP  EQU   *
*  Process data for
*  month number
*  which is in R3

            AH R3,=H'1'     ←  Increment month
                               number
            CH R3,=H'12'    ←  Have you
                                processed all
                               the months
            BNH MTHLOOP     ←  No
```

Branch on Count

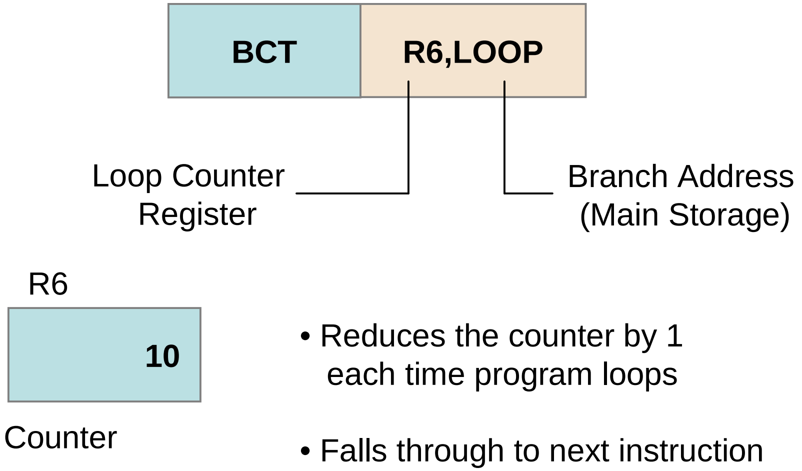
What is Branch on Count?

The Branch on Count (BCT) instruction is designed to handle iterative looping.

One is subtracted from the first operand register and the result is saved in the register. If the result is zero, then normal instruction sequencing occurs at the instruction, following BCT. If the result is non-zero, the main storage address specified by the second operand replaces the next instruction address in the PSW.

If second operand is 0 (zero), no branching occurs and execution continues with the next instruction.

Branch on Count (BCT):
RX Format - Register and Indexed Storage



- Reduces the counter by 1 each time program loops
- Falls through to next instruction when counter reaches 0

CC Setting - Condition Code is Unchanged

Continued…

Branch on Count (cont'd)

How is a BCT used?

Using BCT involves the following steps:

1. Immediately before the body of the loop, load a General Purpose Register (GPR) with the number of times the loop needs to be executed
2. Code the body of the loop, with the last instruction being a BCT, using the initialized register and the address at the start of the loop

The code on the right shows how monthly processing is repeated 12 times using BCT.

```
                LH  R3,=H'12' ←  Loop count
                                 for 12 months
MTHLOOP  EQU *
         *
                                Process data

                                for Month 13-

                                (value in R3)
         *
         BCT  R3,MTHLOOP
```

Continued…

Branch on Count (cont'd)

What happens when BCT does not branch?

The fact that BCT does not branch when its second operand is 0, allows it to be used as a way to subtract one from a register with no literal value needed.

That means, instead of using the following code:

```
SH R7,=H’1’
```

Use:

```
BCTR R7,0
```

The only difference between these two codes is that the BCTR cannot cause overflow and it will not set the condition code.

```
SH     R7,=H'1'
```

Code subtracts the Literal Value 1 from the contents of R7, (requires 2+4=6 bytes)

```
BCTR   R7, 0
```

Code does not branch when the 2nd Operand is 0, allowing it to subtract 1 from R7, (requires 2 bytes)

Unit: Looping Topic: Address Arithmetic

Address Arithmetic

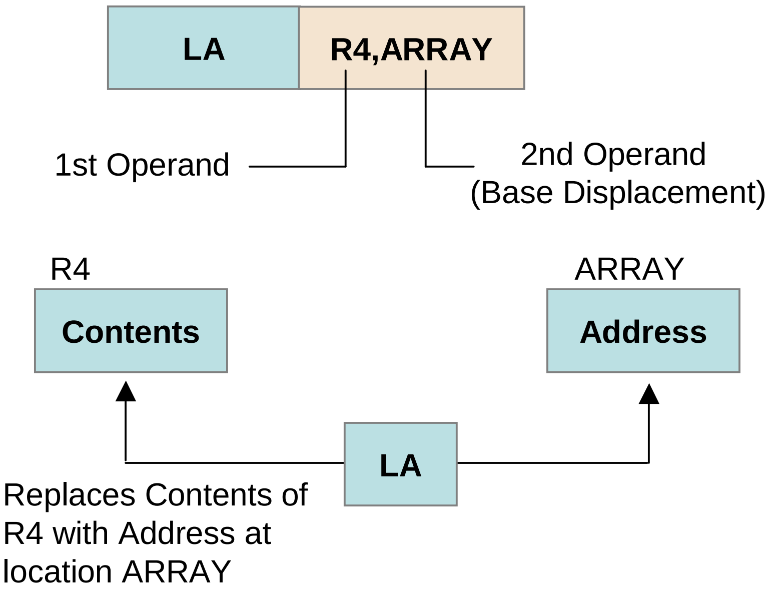
While processing arrays in a loop, each iteration of the loop processes a different element of the array. The Assembler programmer must adjust the address of the current element each time through the loop.

The process of manipulating the addresses of data is called address arithmetic.

The most important instruction used in writing address arithmetic is the Load Address (LA) instruction.

The address of the second operand is placed in the first operand location. The condition code is unaltered.

Load Address (LA):
RX Format - Register and Indexed Storage



CC Setting - Condition Code is Unchanged

Load Address Instruction

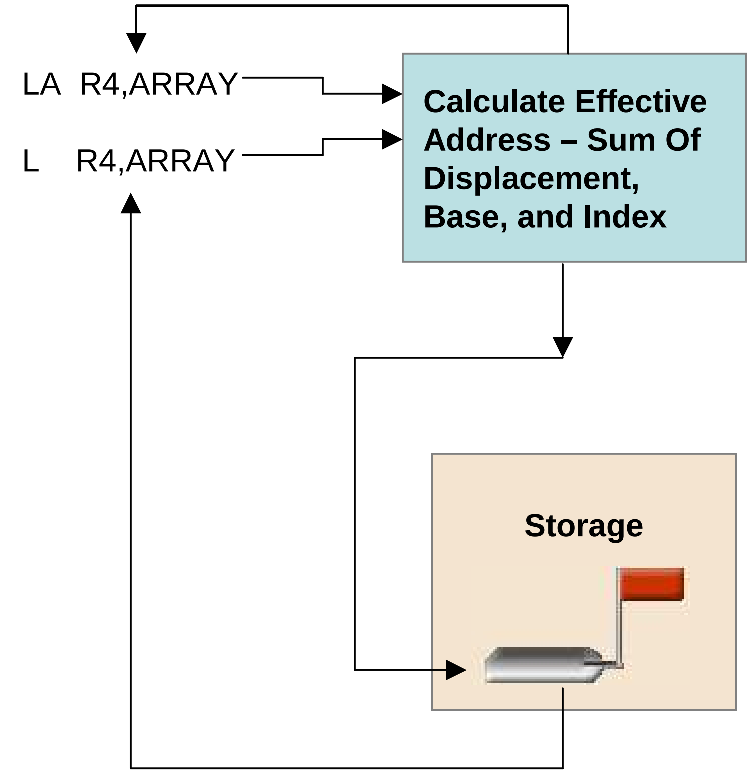
What is the difference between Load and Load Address?

In both cases, the effective address of the second operand is calculated by adding the displacement to the sum of the contents of the base, and index registers. But in the case of Load Address, the effective address is placed in the GPR specified by the first register. For Load, the contents of the effective address are placed into the GPR, specified by the first operand.

Load makes a storage reference, while Load Address does not.

Example: LA R4,ARRAY

The address of the storage location ARRAY is placed into R4.


LA R4,ARRAY
L R4,ARRAY
Calculate Effective Address – Sum Of Displacement, Base, and Index
Storage

Contents Of Effective Address In Storage

Continued…

Load Address Instruction (cont'd)

How effective addresses are formed?

For an RX format instruction, like Load Address, adding the displacement, contents of the base register, and the contents of the index register, forms the effective address.

If either or both of the base and index registers specify R0, then that component is not used in address formation.

LA	**R3, 10**	No Base or Index
LA	**R3, 10(,R4)**	No Index
LA	**R3, 10(R4)**	No Base
LA	**R3,10(R4,R5)**	Both Base and Index

Address Arithmetic

What does address arithmetic deal with?

Address arithmetic deals with storage addresses, which are non-negative integers less than the maximum storage size.

In BC mode, the address arithmetic deals with numbers in the range of 0- 16,777,215.

In EC mode, the maximum number in address arithmetic is $2^{31} – 1$

Continued…

Address Arithmetic (cont'd)

What are the uses of load address?

Load address can be used for the following purposes:

- It can be used for placing the address of a named storage area into a register
- It can be used to initialize a register

Example:

To place the value 10 into R5, instead of using: LH R5,=H'10'
Use, LA R5,10

To add 5 to the contents of R7, instead of using: AH R7,=H'5'
Use, LA R7,5(,R7)

Address Arithmetic: Example

In the example, the effective address is formed by adding the basic contents of the base register R7 and the displacement 5.

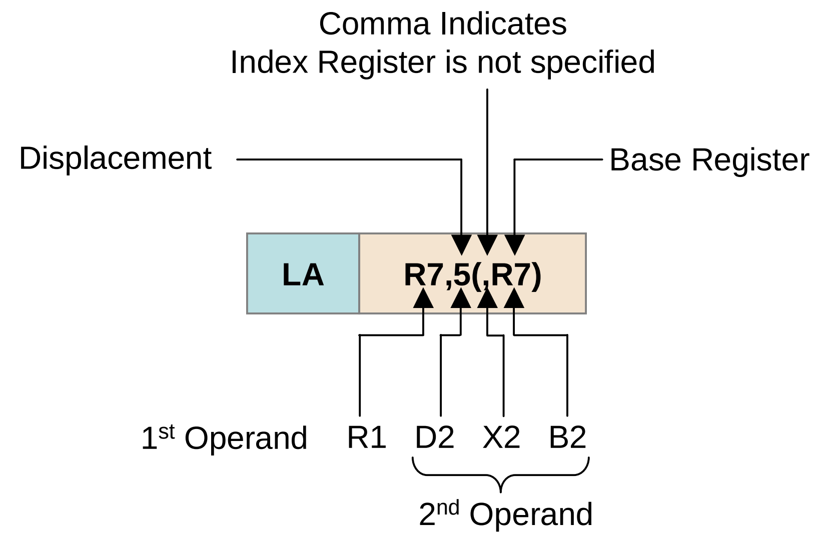
The result is placed in the first operand R7.

The effect is to add 5 to R7.

What are the limitations of address arithmetic?

Following are the limitations of address arithmetic:

- Displacements must be in the range 0-4095
- The arithmetic must deal with non-negative integers less than the storage size, for address arithmetic to work



Basic Instruction Format:
OP R1,D2,(X2,B2)

Defining Arrays

An array is a repetitive data structure, consisting of multiple elements of some base type.

In the example shown to record a a company’s gross sales for each day in January, an array of 31 fullwords, or 31 packed decimal numbers have been used.

Fixed-Point Array:

```
FDSALES      DS     31F
```

Array of 31 Fullwords

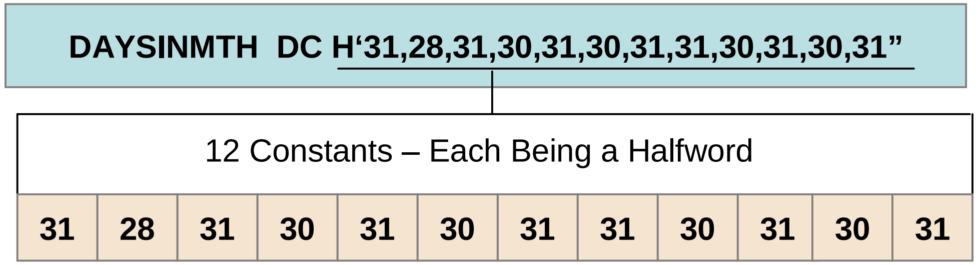
Packed Decimal:

```
PDSALES      DS     31PL4
```

31 Packed Decimal Fields of Length 4

Continued…

Defining Arrays (cont’d)



DAYSINMTH DC H‘31,28,31,30,31,30,31,31,30,31,30,31”

12 Constants – Each Being a Halfword

31	**28**	**31**	**30**	**31**	**30**	**31**	**31**	**30**	**31**	**30**	**31**

Main Storage – 12 Elements

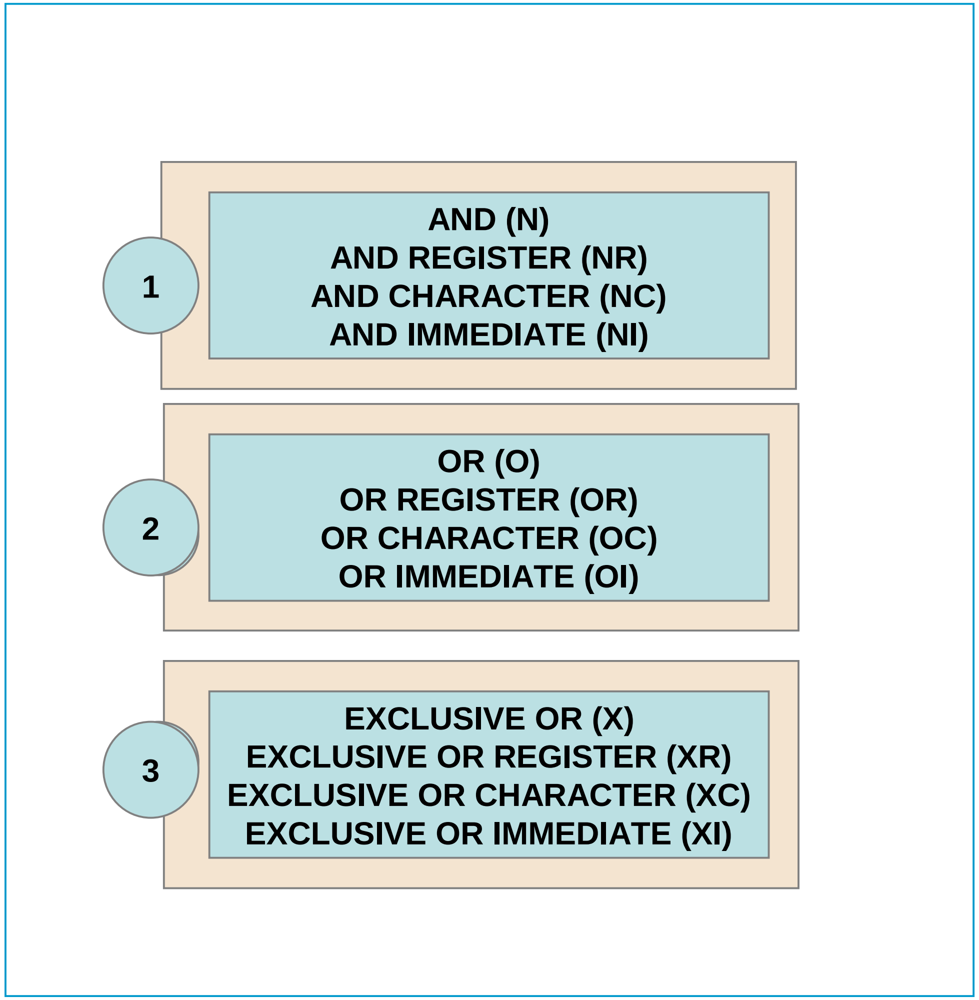
The example shows how to create an array of constants. It shows an array to hold the number of days in each month of a common (not a leap) year.

Here a duplication factor has not been specified. Rather, by defining twelve constants, the Assembler has been told that the array has 12 elements.

Boolean Algebra

Boolean algebra is an algebra that deals with the binary values of zero and one. In this algebra, 0 often represents false and 1 represents true.

The internal operation of computers is based on Boolean logic, and most computer architectures provide some Boolean instructions in their instruction set.



Continued…

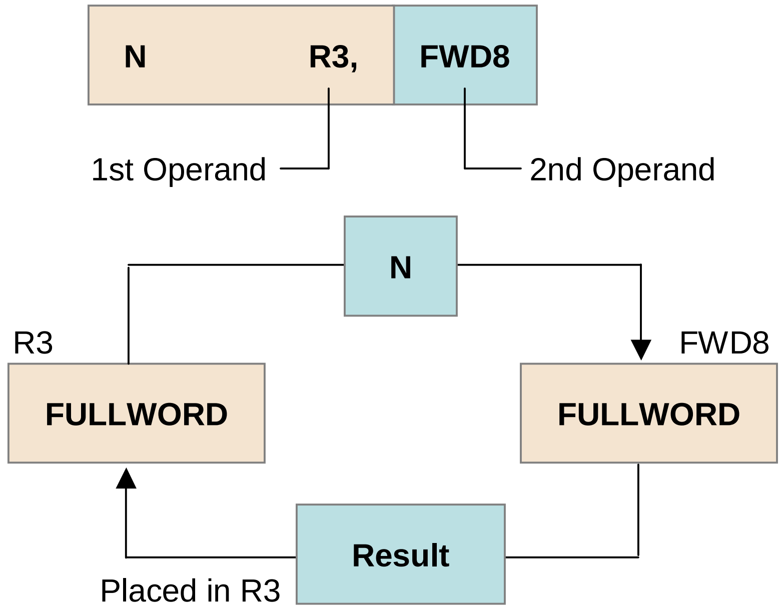
Machine Instructions with Boolean AND

AND (N) Instruction

Here is an example of AND (N) instruction, which is an RX format instruction.

Here, the 32 bits of the first operand (in a GPR) are bit-wise ANDed with the 32 bits of the second operand, which is a fullword in main storage. The result replaces the first operand.

AND (N):
RX Format – Register and Indexed Storage



CC Setting: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

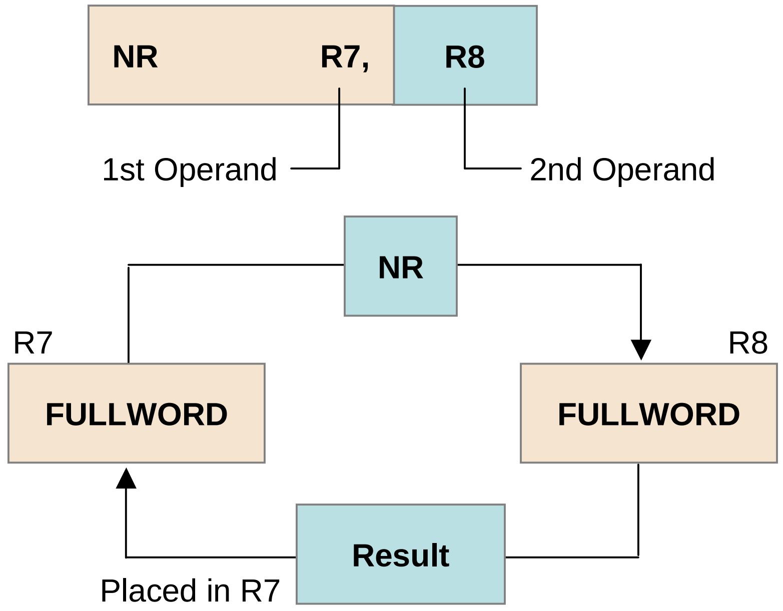
Machine Instructions with Boolean AND (cont'd)

AND (NR) Instruction

Here is an example of AND (NR) instruction.

In this RR format instruction, the 32 bits in the first operand register are bit wise ANDed with the 32 bits in the second operand register. The result replaces the first operand.

AND (NR):
RR Format – Register and Register



CC Setting: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

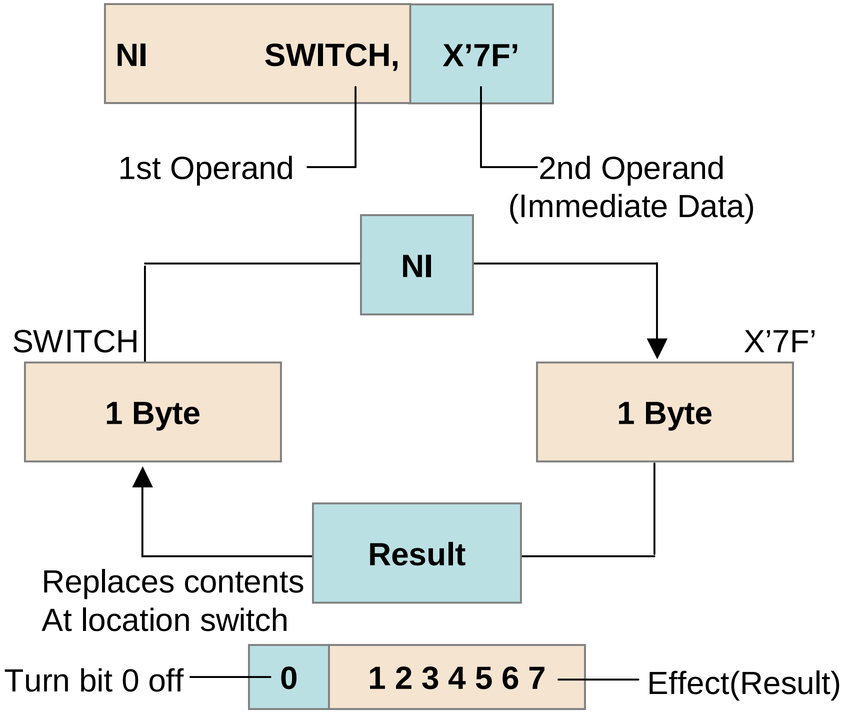
Machine Instructions with Boolean AND (cont'd)

AND (NI) Instruction

The AND (NI) instruction is shown in the example.

In this SI format instruction, the eight bits of the first operand are bit-wise ANDed with the 8 bits of the immediate operand, and the result replaces the first operand.

AND (NI):
SI Format – Storage and Immediate Data



CC Setting: 0 - Result is all zero bits
1 - Result is not all zero bits

Continued…

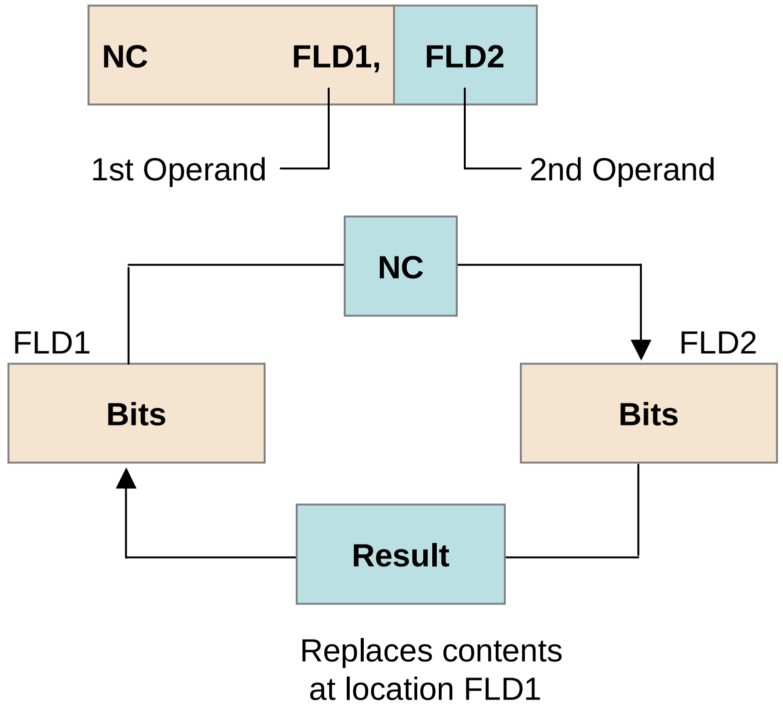
Machine Instructions with Boolean AND (cont'd)

AND (NC) Instruction

The AND (NC) instruction is shown in the example.

In this SS format instruction, the bits of the first operand are bit-wise ANDed with the bits of the second operand, with the result replacing the first operand.

AND (NC):
SS Format – Storage and Storage



CC Setting: 0 - Result is all zero bits
1 – Result is not all zero bits

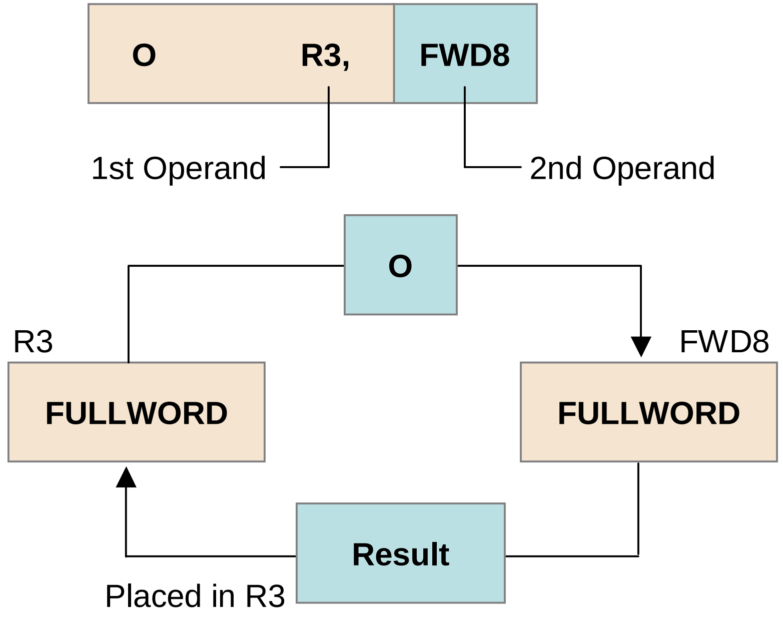
Machine Instructions with Boolean OR

OR (O) Instruction

The OR (O) instruction is shown in the example.

In this RX format instruction, the 32 bits of the first operand (in a GPR) are bit-wise ORed with the 32 bits of the second operand, witch is a fullword in main storage. The result replaces the first operand.

OR (O):
RX Format – Register and Indexed Storage



CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

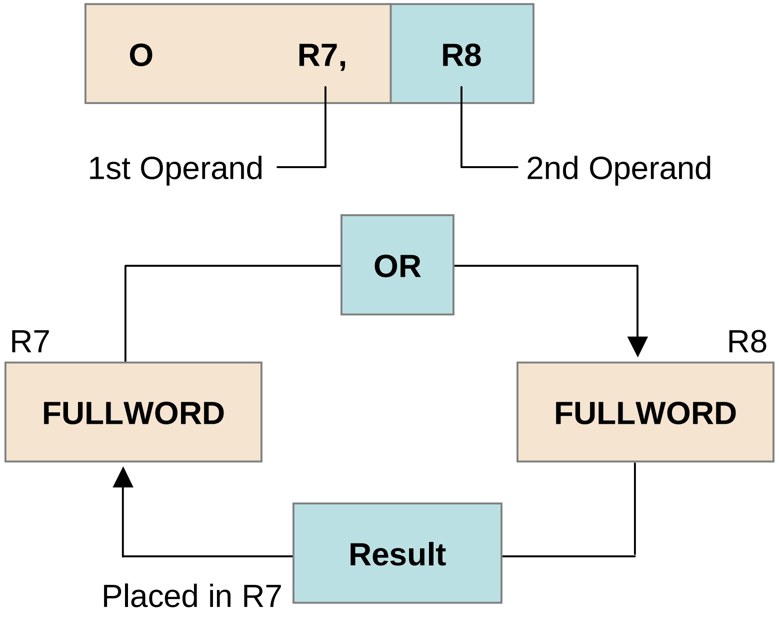
Machine Instructions with Boolean OR (cont'd)

OR (OR) Instruction

The OR (OR) instruction is shown in the example.

In this RR format instruction, the 32 bits of the first operand register are bit-wise ORed with the 32 bits in the second operand register. The result replaces the first operand.

OR (OR):
RR Format – Register and Register



CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

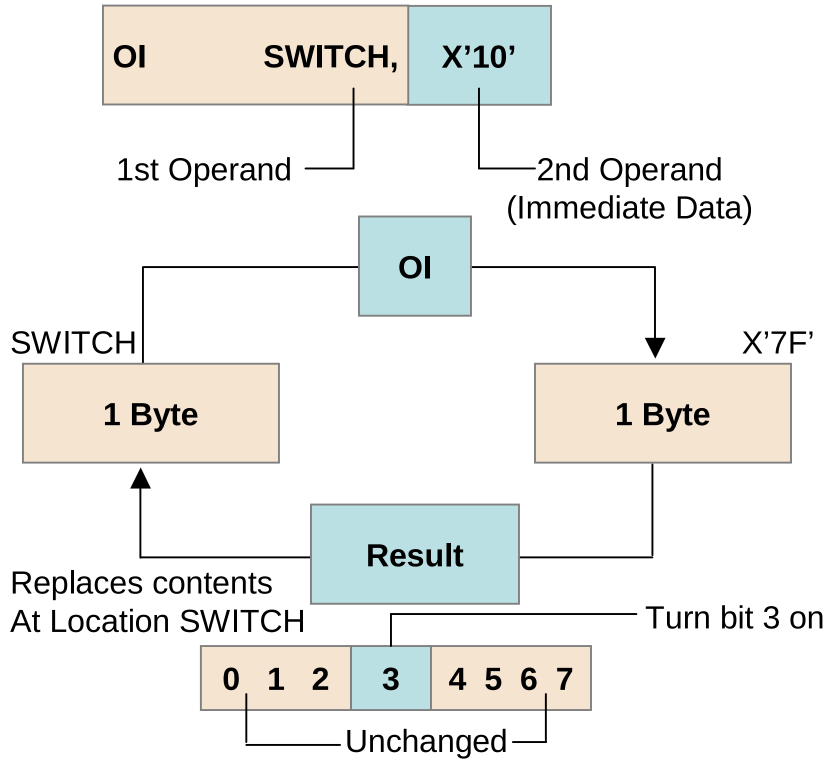
Machine Instructions with Boolean OR (cont'd)

OR (OI) Instruction

The OR (OI) instruction is shown in the example.

In this SI format instruction, the eight bits of the first operand are bit-wise ORed with the 8 bits of the immediate operand, the result replaces the first operand.

OR (OI):
SI Format – Storage and Immediate Data



CC Settings: 0 - Result is all zero bits
1 - Result is not all zero bits

Continued…

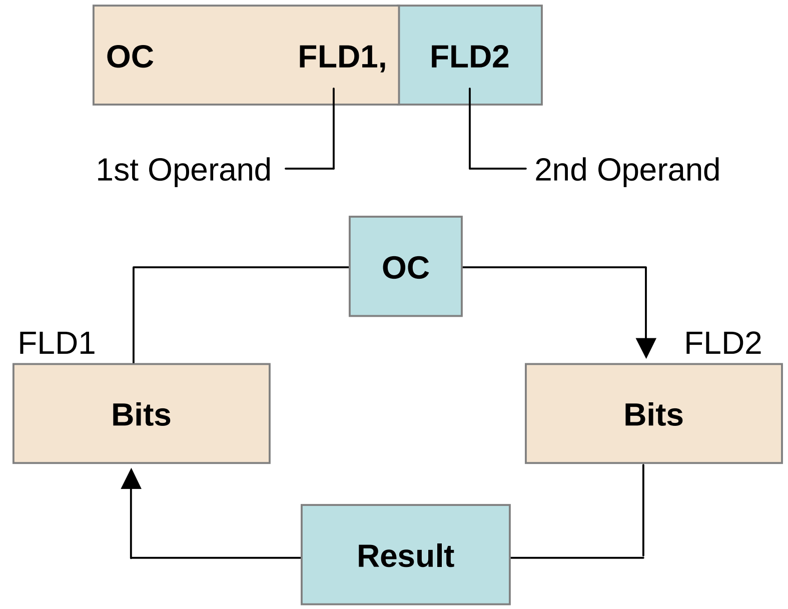
Machine Instructions with Boolean OR (cont'd)

OR (OC) Instruction

The OR (OC) instruction is shown in the example.

In this SS format instruction, the bits of the first operand are bit-wise ORed with the bits of the second operand, with the result replacing the first operand.

OR (OC):
SS (1 length) Format – Storage and Storage



Replaces Contents
at Location FLD1

CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

Machine Instructions with Boolean OR (cont'd)

OR (OC) Instruction

One use of the OC instruction is to convert text in mixed case to all upper case.

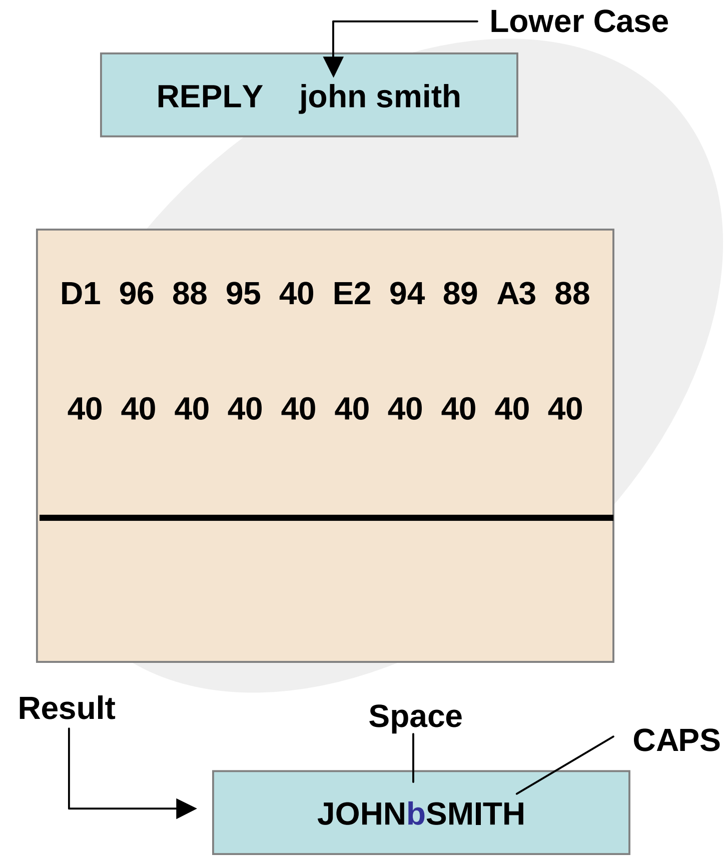
The technique of using OC to change to uppercase works because the EBCDIC codes for the lower case letters is almost identical to the codes for the uppercase.

The example shows a user reply into a 20-byte field called REPLY. To convert the field to uppercase, following code can be used.

```
OC    REPLY,=20X'40'
```

X'40' is the EBCDIC code for blank, so it can be coded as:

```
OC    REPLY,=CL20' '
```



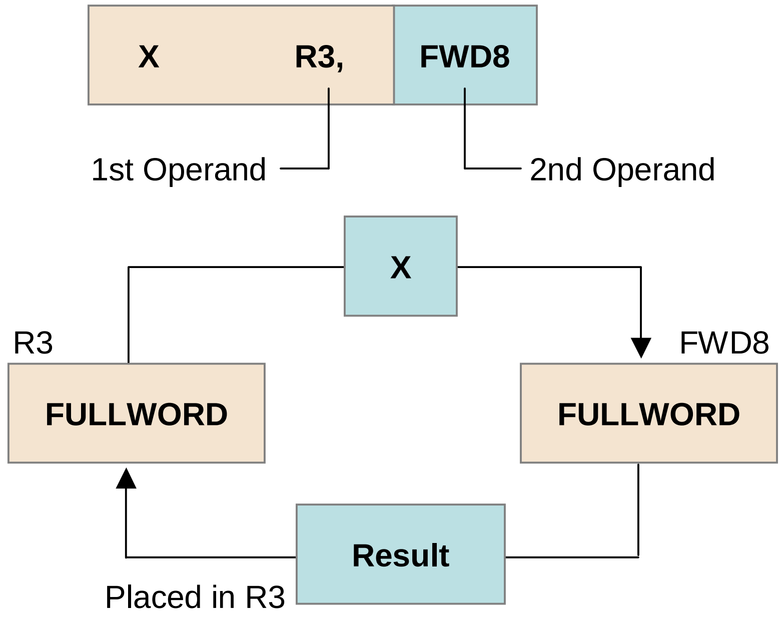
Machine Instructions with Boolean Exclusive OR

EXCLUSIVE OR (X) Instruction

The EXCLUSIVE OR (X) instruction is shown in the example.

In this RX format instruction, the 32 bits of the first operand (in a GPR) are bit-wise XORed with the 32 bits of the second operand, which is a fullword in main storage. The result replaces the first operand.

EXCLUSIVE OR (X):
RX Format – Register and Indexed Storage



CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

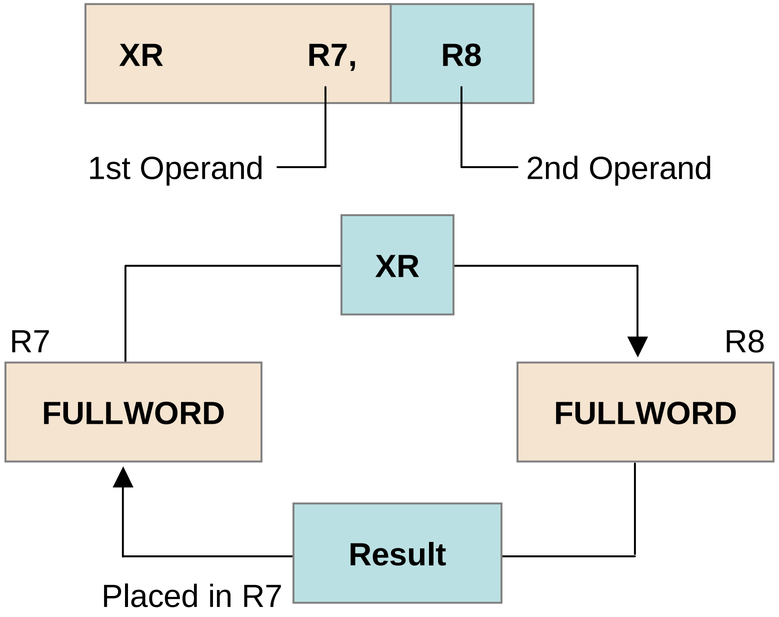
Machine Instructions with Boolean Exclusive OR (cont'd)

EXCLUSIVE OR (XR) Instruction

The EXCLUSIVE OR (XR) instruction is shown in the example.

In this RR format instruction, the 32 bits of the first operand register are bit-wise XORed with the 32 bits in the second operand register. The result replaces the first operand.

EXCLUSIVE OR (XR):
RR Format – Register and Register



CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

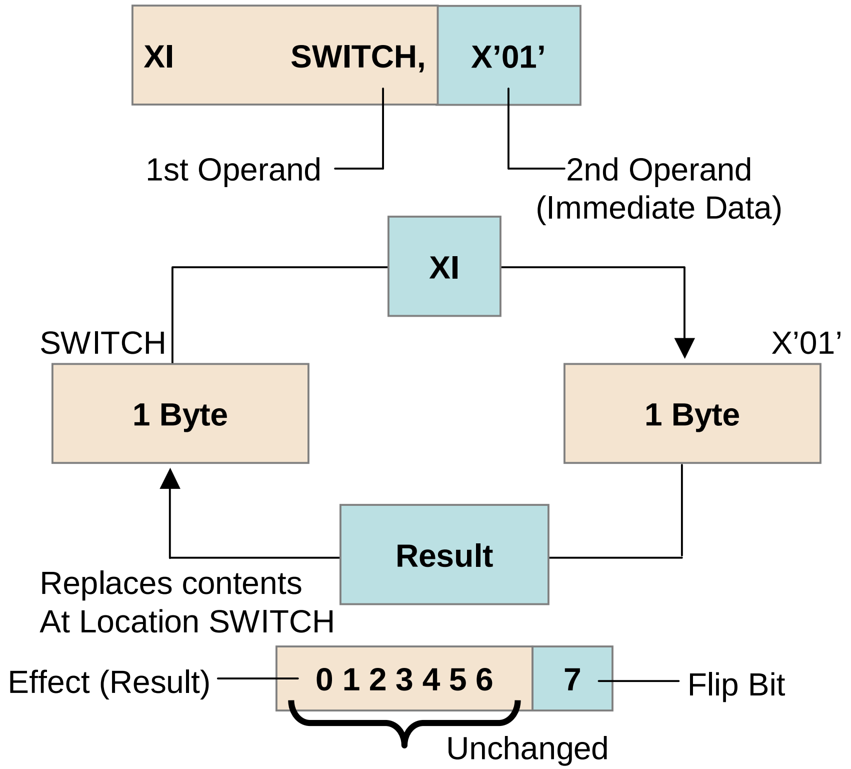
Machine Instructions with Boolean Exclusive OR (cont'd)

EXCLUSIVE OR (XI) Instruction

The EXCLUSIVE OR (XI) instruction is shown in the example.

In this SI format instruction, the eight bits of the first operand are bit-wise XORed with the 8 bits of the immediate operand, the result replaces the first operand.

EXCLUSIVE OR (XI):
SI Format – Storage and Immediate Data



CC Settings: 0 - Result is all zero bits
1 - Result is not all zero bits

Continued…

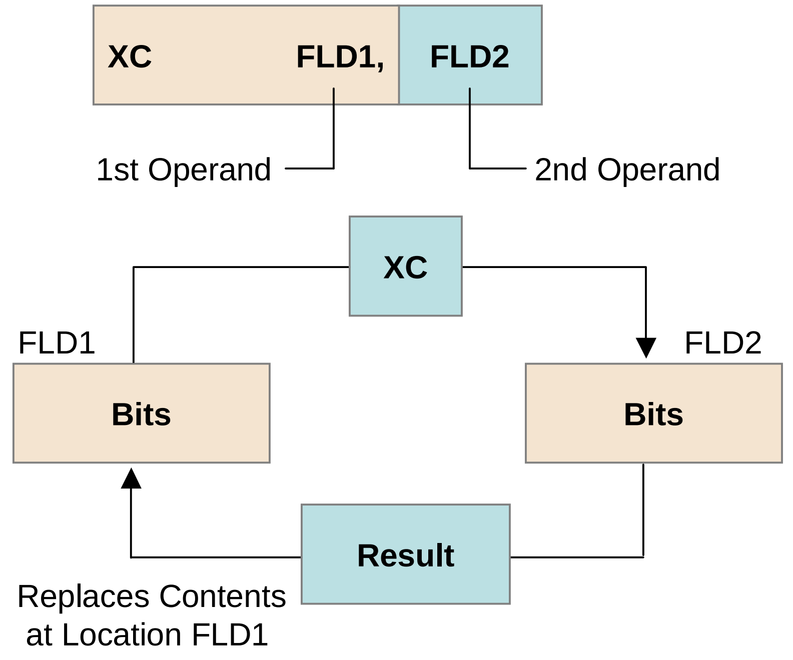
Machine Instructions with Boolean Exclusive OR (cont'd)

EXCLUSIVE OR (XC) Instruction

The EXCLUSIVE OR (XC) instruction is shown in the example.

In this SS format instruction, the bits of the first operand are bit-wise XORed with the bits of the second operand, with the result replacing the first operand.

EXCLUSIVE OR (XR):
SS Format – Storage and Storage



CC Settings: 0 - Result is all zero bits
1 – Result is not all zero bits

Continued…

Machine Instructions with Boolean Exclusive OR (cont'd)

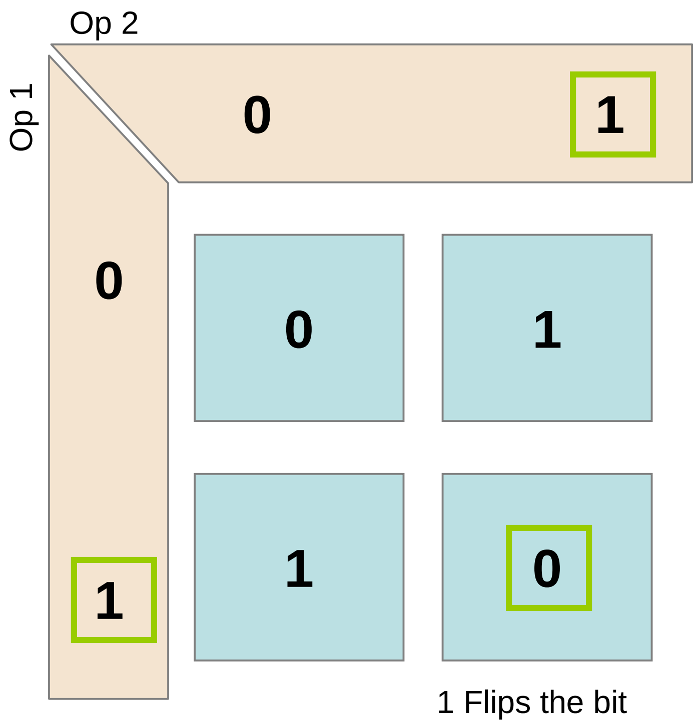
EXCLUSIVE OR (XC) Instruction

There are two important applications of EXCLUSIVE OR. They are:

- To set a field to binary zeros
- To write code to sort data

To Set A Field To Binary Zeros

The first application is shown in the truth table for Exclusive OR. It shows that after a field is Exclusive ORed with itself, the result will be all zeros.



	0	1
0	0	1
1	1	0

1 Flips the bit
0 retains the bit

0 XOR 0 = 0

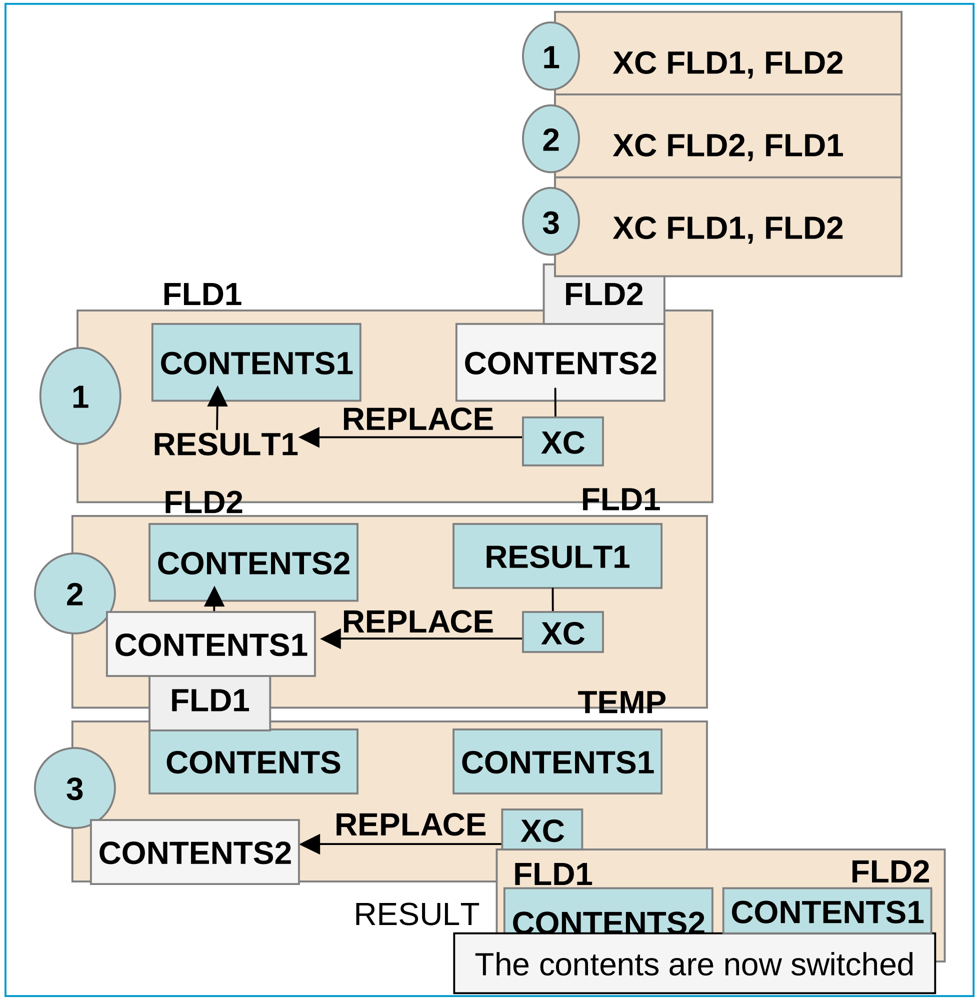
1 XOR 1 = 0

Continued…

Machine Instructions with Boolean Exclusive OR (cont'd)

Using the XC instruction, it is possible to exchange values with the same number of instructions, but with no intermediate data area, thus saving storage.

The code is shown in the example.



Continued…

Shifting Instructions

Shifting instructions help in handling applications that require altering bits of data in registers and storage. Shifting instructions move the content of GPRs to the left or the right, by a specified number of bits.

The amount of data shifted can be 32 bits in a single GPR or 64 bits in an even-odd register pair. The data can be interpreted as an unsigned logical quantity (logical shift) or as a signed arithmetic quantity (arithmetic shift).

All combinations of the 3 types of shifting are allowed, producing 2 x 2 x 2 or 8 shift instructions.

Logical Instructions:

Arithmetic Shift:	
Shift Left Single (Algebraic)	**(SLA)**
Shift Left Double (Algebraic)	**(SLDA)**
Shift Right Single (Algebraic)	**(SRA)**
Shift Right Double (Algebraic)	**(SRDA)**

Logical Shaft:	
Shift Left Single Logical	**(SLL)**
Shift Left Double Logical	**(SLDL)**
Shift Right Single Logical	**(SRL)**
Shift Right Double Logical	**(SRDL)**

Types of Shifting

Direction	Size	Type	Instruction
right	single	logical	SRL
right	single	arithmetic	SRA
right	double	logical	SRDL
right	double	arithmetic	SRDA
Left	single	logical	SLL
Left	single	arithmetic	SLA
Left	double	logical	SLDL
Left	double	arithmetic	SLDA

There are eight shifting instructions in total. These instructions shift the contents of GPRs on a bit-wise basis. Registers contents can be shifted left or right.

The eight instructions are shown here.

Continued…

Types of Shifting (cont'd)

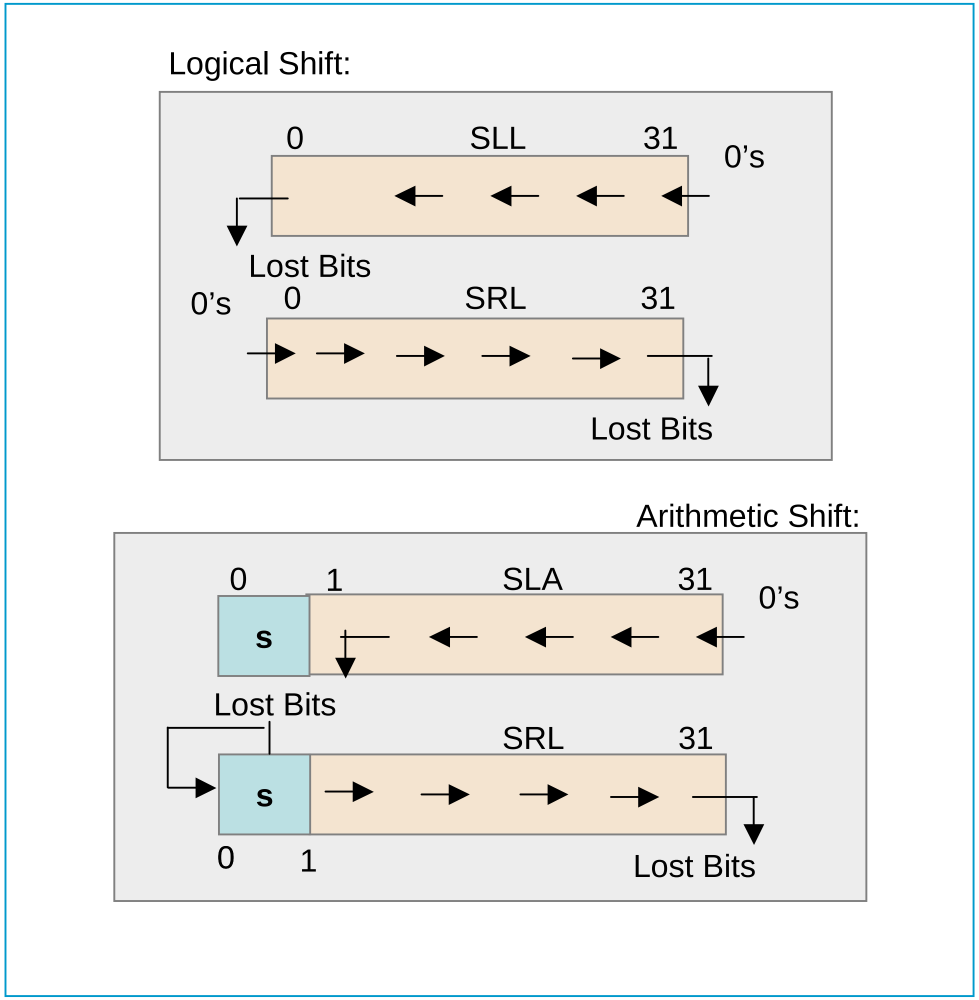
What happens during a logical shift?

In a logical shift, all of the bits participate. Bits that are shifted out of the register at the right end for a right shift and at the left end for a left shift disappear.

Vacated bit positions are filled with zero bits. The condition code is not changed in logical shifts.

What happens during an arithmetic shift?

In an arithmetic shift, only the 31 or 63 bits to the right of the sign bit participate. Bits shifted out of the register disappear. In a left arithmetic shift, vacated bit positions are filled with zero bits. In a right arithmetic shift, bits equal to the sign bit replace vacated bit positions. The condition code is set in a similar way to arithmetic instructions. An overflow can occur in a left shift, when a significant bit is shifted out of the register.



Continued…

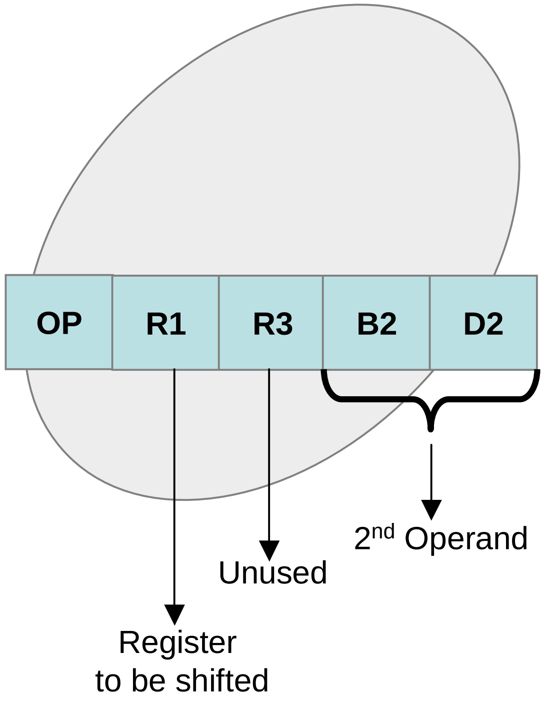
Types of Shifting (cont'd)

The shift instructions are all RS type. The R3 field is not used in shift instructions, and is ignored in the Machine Language format. The second operand does not specify a field in main storage, but it is used to represent the magnitude of the shift.

The address arithmetic is done, and the rightmost 6 bits of that result specifies how many bit positions to shift the first operand. Normally in Assembler language source statements, this field is specified as a displacement, indicating the number of bit positions to move the register contents.

A single shift instruction can specify any register as the first operand. A double shift instruction must specify an even numbered register, implying that particular register and the next higher register.

Shift Instructions:
RS Format – Register and Storage



Logical Shifting Instructions

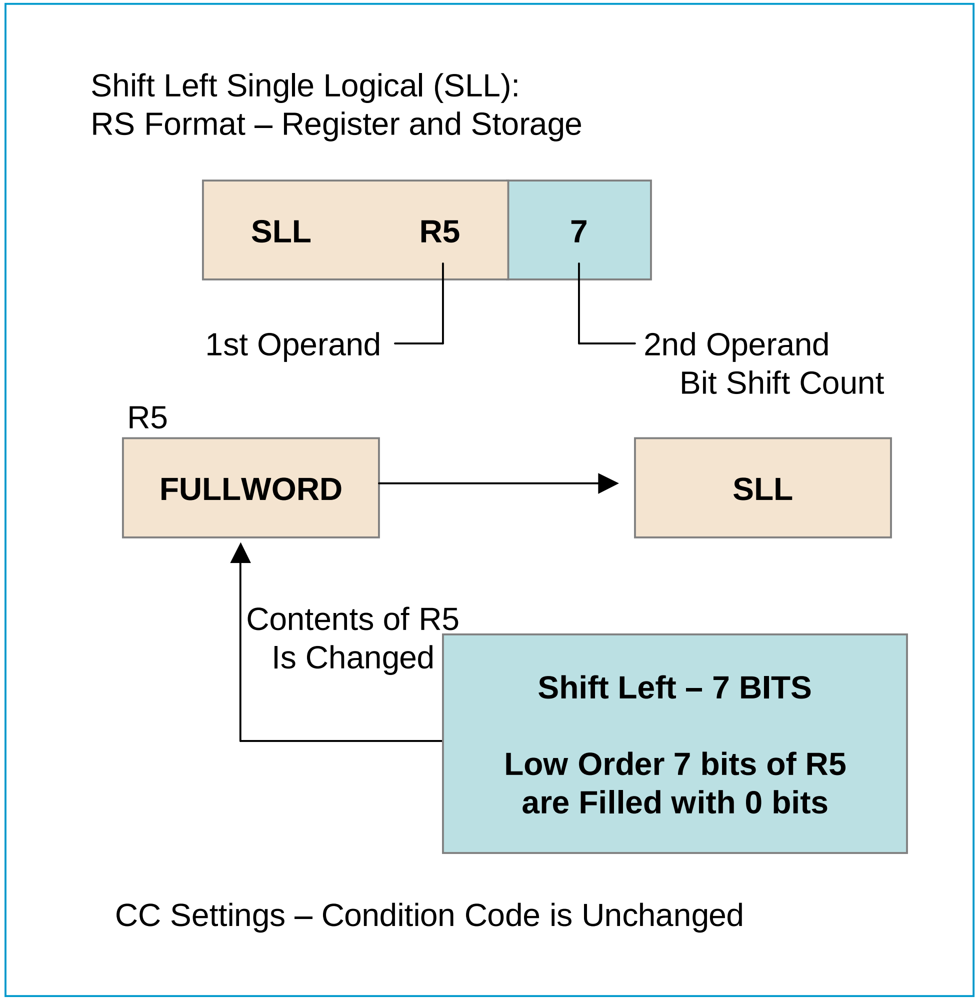
Shift Left Single Logical Instruction

The Shift Left Single Logical (SLL) instruction is shown in the example.

With the SLL, an RS format instruction, the 32 bits in the first operand shift left by the amount specified by the rightmost 6 bits of the effective address of the second operand. Zero bits replace the displaced bits on the right.

Example:

SLL R5,7



Continued…

Logical Shifting Instructions (cont’d)

Shift Right Single Logical Instruction

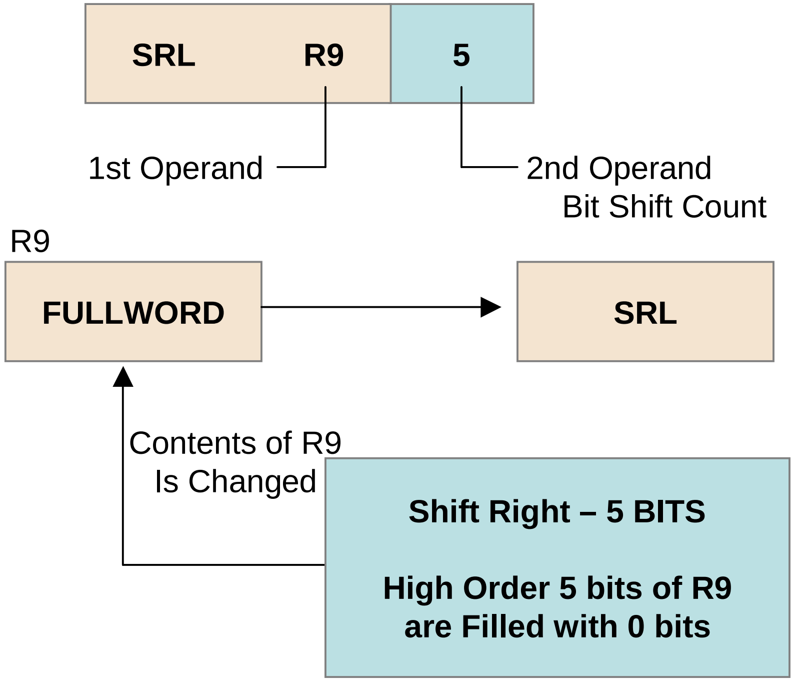
The Shift Right Single Logical (SRL) instruction is shown in the example.

With the SRL, an RS format instruction, the 32 bits in the first operand shift right by the amount specified by the rightmost 6 bits of the effective address of the second operand. Zero bits replace the displaced bits on the left.

Example:

SRL R9,5

Shift Right Single Logical (SRL):
RS Format – Register Storage



CC Settings – Condition Code is Unchanged

Continued…

Logical Shifting Instructions (cont'd)

Shift Left Double Logical Instruction

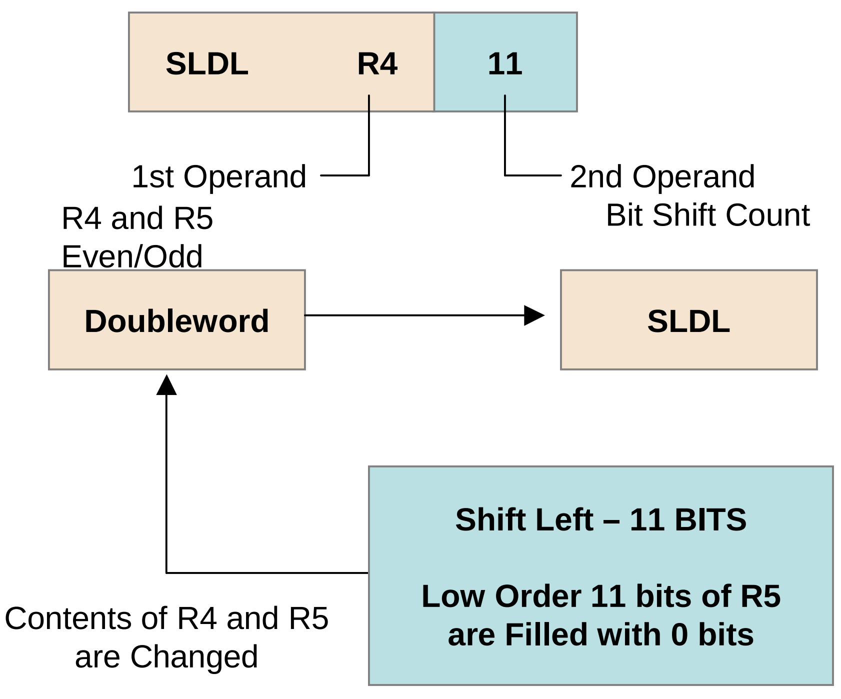
The Shift Left Double Logical (SLDL) instruction is shown in the example.

The SLDL, an RS format instruction, is similar to the SLL instruction, except that it uses paired registers. The instruction shifts 64 bits in the first operand register, and the next higher register to the left. The magnitude of the shift is specified, by the rightmost 6 bits of the effective address of the second operand. Zero bits replace the displaced bits on the right.

Example:

SLDL R4,11

Shift Left Double Logical (SLDL):
RS Format – Register and Storage



CC Settings – Condition Code is Unchanged

Continued…

Logical Shifting Instructions (cont'd)

Shift Right Double Logical Instruction

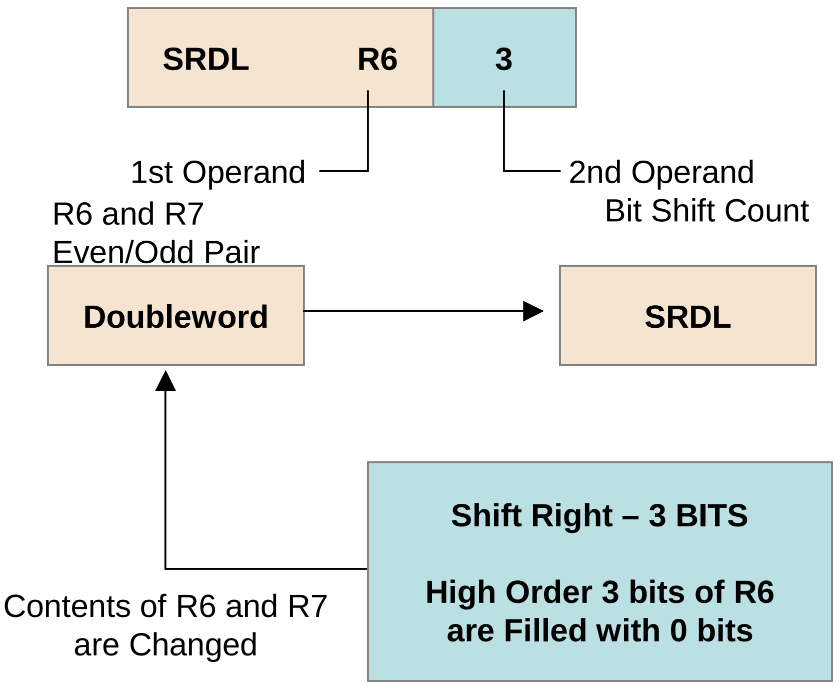
The Shift Right Double Logical (SRDL) instruction is shown in the example.

The SRDL, an RS format instruction, is similar to the SRL instruction. This instruction, however, shifts 64 bits in the first operand register, and to the next higher register to the right. The rightmost 6 bits of the effective address of the second operand specify the magnitude of the shift. Zero bits replace the displaced bits on the left.

Example:

SRDL R6,3

Shift Right Double Logical (SRDL):
RS Format – Register Storage



CC Settings – Condition Code is Unchanged

Continued…

Logical Shifting Instructions (cont'd)

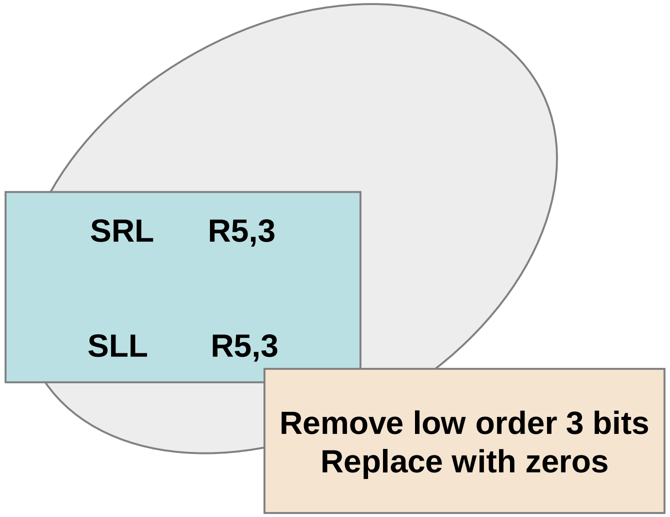
There are many applications of the logical shift instructions that involve bit manipulation and related application.

If R5 contains a storage address, to determine the first address at or below the address in R5, that is, a doubleword boundary.

To find the boundary, the low order 3 bits of the address should be replaced by zeros.

The example shows how it is done using SRL and SLL instructions.

Doubleword Boundaries:



```
SRL     R5,3

SLL     R5,3
```

Remove low order 3 bits
Replace with zeros

Arithmetic Shifting Instructions

Shift Left Single (Algebraic) Instruction

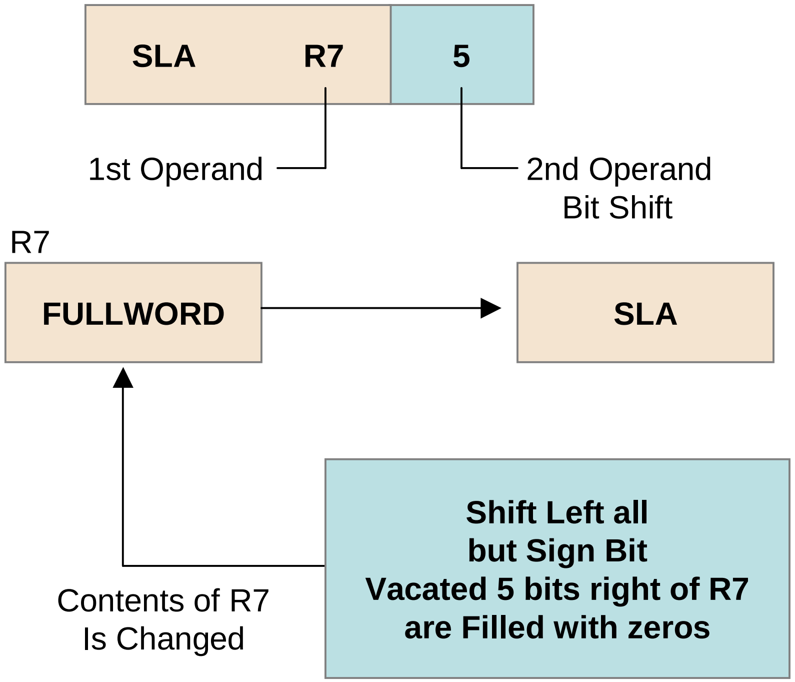
The Shift Left Single (Algebraic) (SLA) instruction is shown in the example.

SLA, an RS format instruction, shifts numeric bits of the signed first operand to the left. The rightmost 6 bits of the effective address of the second operand, specify the magnitude of the shift. Zero bits replace the displaced bits on the right.

Example:

SLA R7,5

Shift Left Single (SLA):
RS Format – Register and Storage



CC Settings – Arithmetic

Continued…

Arithmetic Shifting Instructions (cont'd)

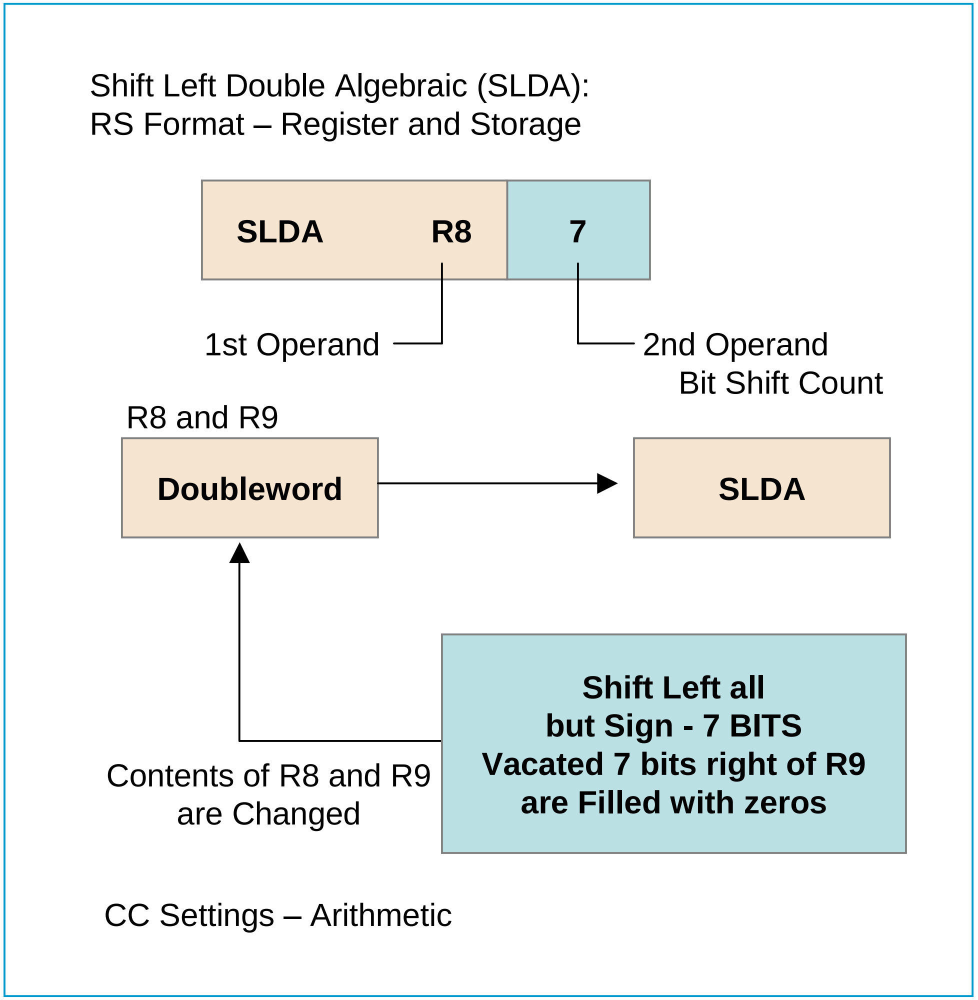
Shift Left Double (Algebraic) (SLDA) Instruction:

The Shift Left Double Algebraic (SLDA) instruction is shown in the example.

The SLDA, an RS format instruction, is similar to SLA, but operates on the data in an even-odd register pair. The 63 bits are shifted left. The magnitude of the shift is specified, by the second operand. Zero bits replace the displaced bits on the right.

Example:

SLDA R8,7


Shift Left Double Algebraic (SLDA):
RS Format – Register and Storage
SLDA
R8
7
1st Operand
2nd Operand
Bit Shift Count
R8 and R9
Doubleword
SLDA
Shift Left all but Sign - 7 BITS
Vacated 7 bits right of R9 are Filled with zeros
Contents of R8 and R9 are Changed
CC Settings – Arithmetic

Continued…

Arithmetic Shifting Instructions (cont'd)

Shift Right Single (Algebraic) Instruction

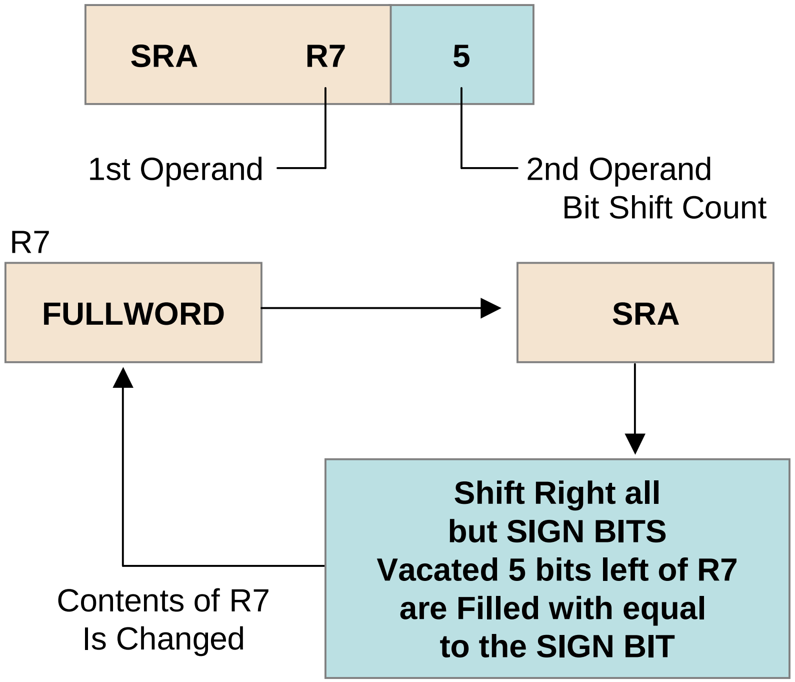
The Shift Right Single (Algebraic) (SRA) instruction is shown in the example.

SRA, an RS format instruction, shifts 31 numeric bits of the signed operand to the right. The rightmost 6 bits of the effective address of the second operand, specify the magnitude of the shift. Bits equal to the sign bit replace the displaced bits on the left.

Example:

SRA R7,5

Shift Right Single Algebraic (SRA):
RS Format – Register and Storage



CC Settings – 0 – Result is zero
1 – Result is less then zero
2 – Result is greater then zero

Continued…

Arithmetic Shifting Instructions (cont'd)

Shift Right Double (Algebraic) Instruction

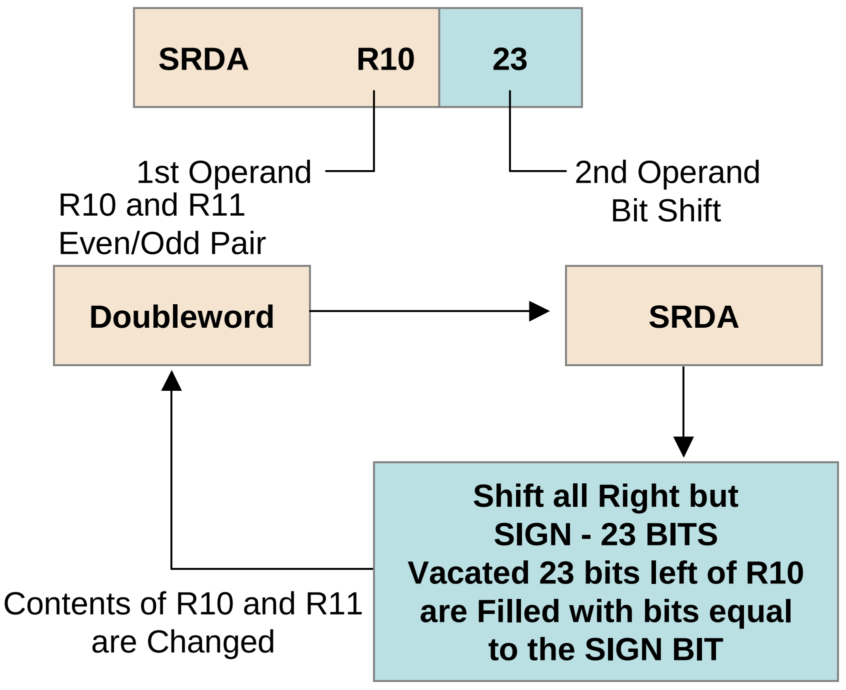
The Shift Right Double (Algebraic) (SRDA) instruction is shown in the graphic.

SRDA, an RS format instruction, is similar to SRA, but operates on the data in an even-odd register pair. The 63 bits are shifted right. The magnitude of the shift is specified by the second operand. Bits equal to the sign bit replace the displaced bits on the left.

Example:

SRDA R10,23

Shift Right Double Algebraic (SRDA):
RS Format – Register and Storage


SRDA
R10
23
1st Operand
R10 and R11
Even/Odd Pair
2nd Operand
Bit Shift
Doubleword
SRDA
Shift all Right but SIGN - 23 BITS Vacated 23 bits left of R10 are Filled with bits equal to the SIGN BIT
Contents of R10 and R11 are Changed

CC Settings – 0 – Result is zero
1 – Result is less then zero
2 – Result is greater then zero

Continued…

Test Under Mask

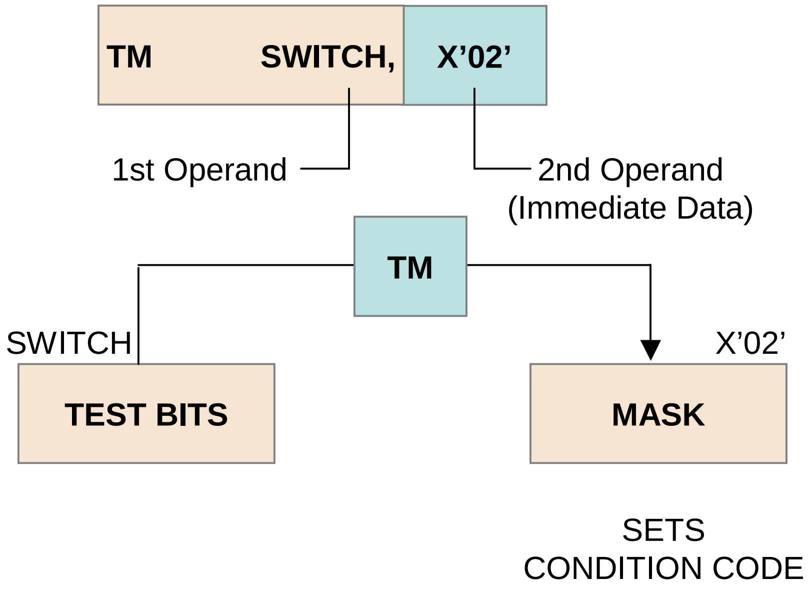
The Test under Mask instruction has been summarized as shown.

In the TM instruction, the immediate operand, which is the second operand, is an 8-bit mask. Bits to be tested in the first operand are specified by placing a 1 in the corresponding mask position. Neither operand is changed, but the condition code is set..

Example:

TM SWITCH,X’02’

TEST UNDER MASK (TM):
SI Format – Register and Immediate Data



TM SWITCH, X’02’

1st Operand

2nd Operand
(Immediate Data)

TM

SWITCH

TEST BITS

X’02’

MASK

SETS
CONDITION CODE

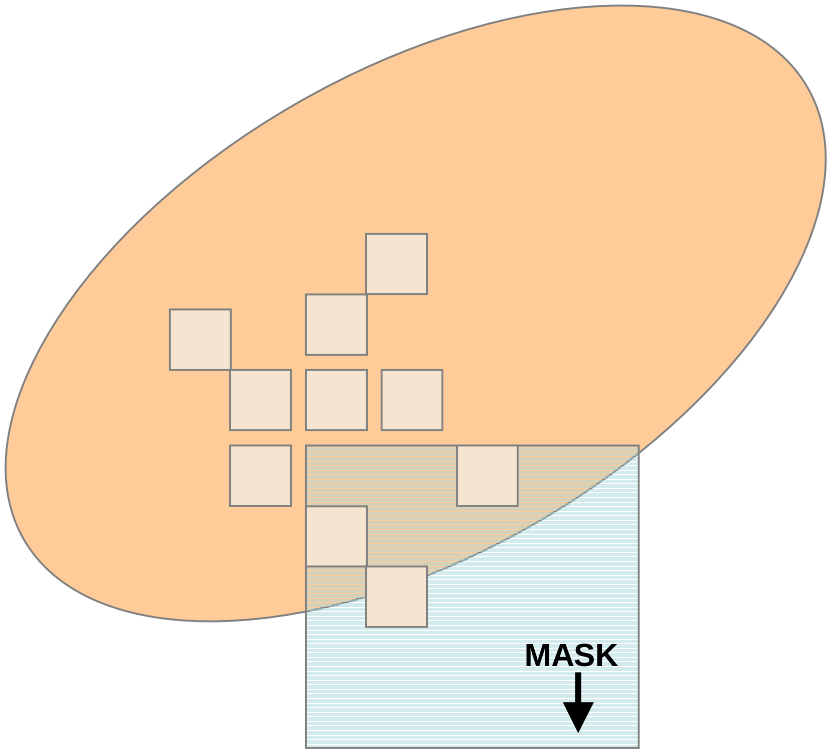
CC Settings – 0 – Mask is zero
or all selected bits are zero
1 – Some selected bits are zeros
and some are ones
2 – All selected bits are ones

Continued…

Test Under Mask (cont'd)

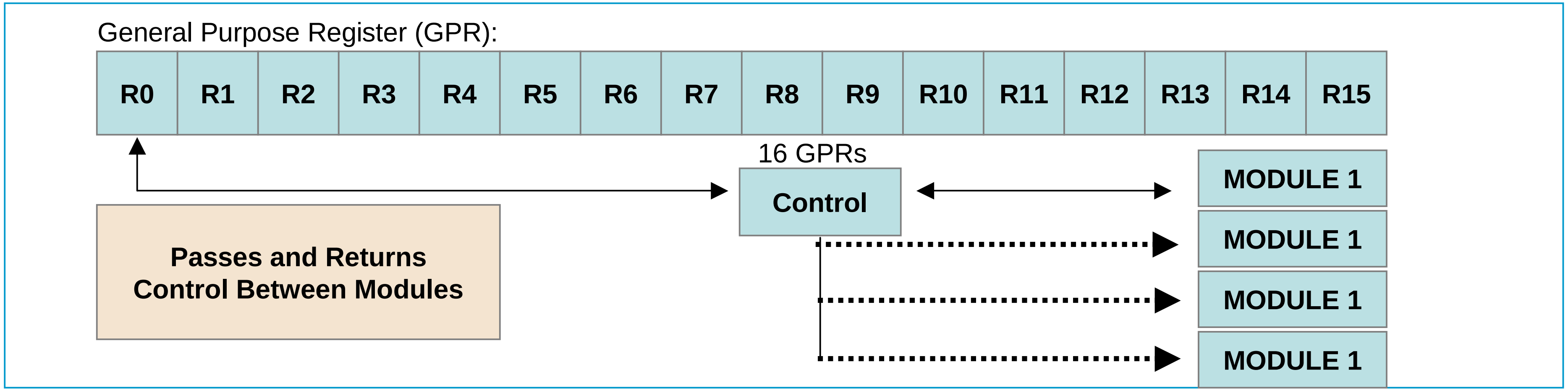
The TM instruction can test a single bit, or multiple bits. A condition code of 1 is only possible if multiple bits are being tested.

The condition code values assigned to the various conditions may seem arbitrary. The reason for these settings is to enable the extended branch mnemonics Branch if Zeros (BZ), Branch if Mixed (BM) and Branch if Ones (BO) to be used.



Specifies Bits
to Test

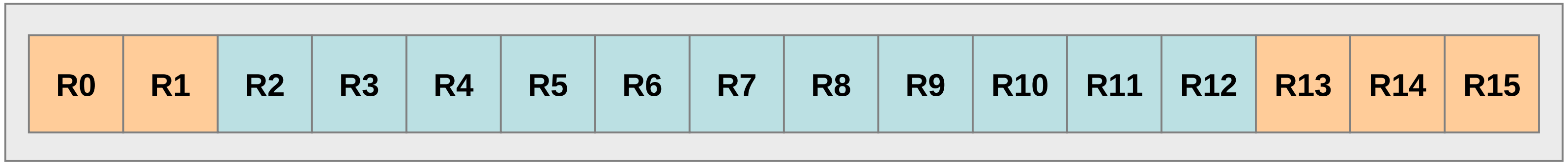
Conventional Use of Registers



In order to provide consistency and ease of linkage between programs written in different languages, the operating system defines certain conventions for passing and returning control and data between modules. These conventions concern the standard use of the General Purpose Registers (GPRs) and the procedure for saving and restoring them, as control passes from module to module.

Continued…

Conventional Use of Registers (cont'd)



R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15

When passing and returning control and data between modules, five of the GPRs (R0, R1, R13, R14 and R15) are used in conventional ways.

These five registers are also used by many of the system macro instructions.

Continued…

Conventional Use of Registers (cont'd)

Here is a list of the five GPRs with their description:

R0 - Returns a single fullword value from a called program to its caller. If the called program is a function, and the type of the function is a fullword or smaller type, then R0 is used to return the function result.

R1- Used to pass the address of a parameter list from a calling program to its called program. The address of the parameter list is passed in R1.

R13- Used to hold the save area address. The area used to save registers is called a saved area, and the address of the save area is held in R13.

R14- Used to hold the return address in the caller when control is passed. When the called module completes processing, it returns control to this address.

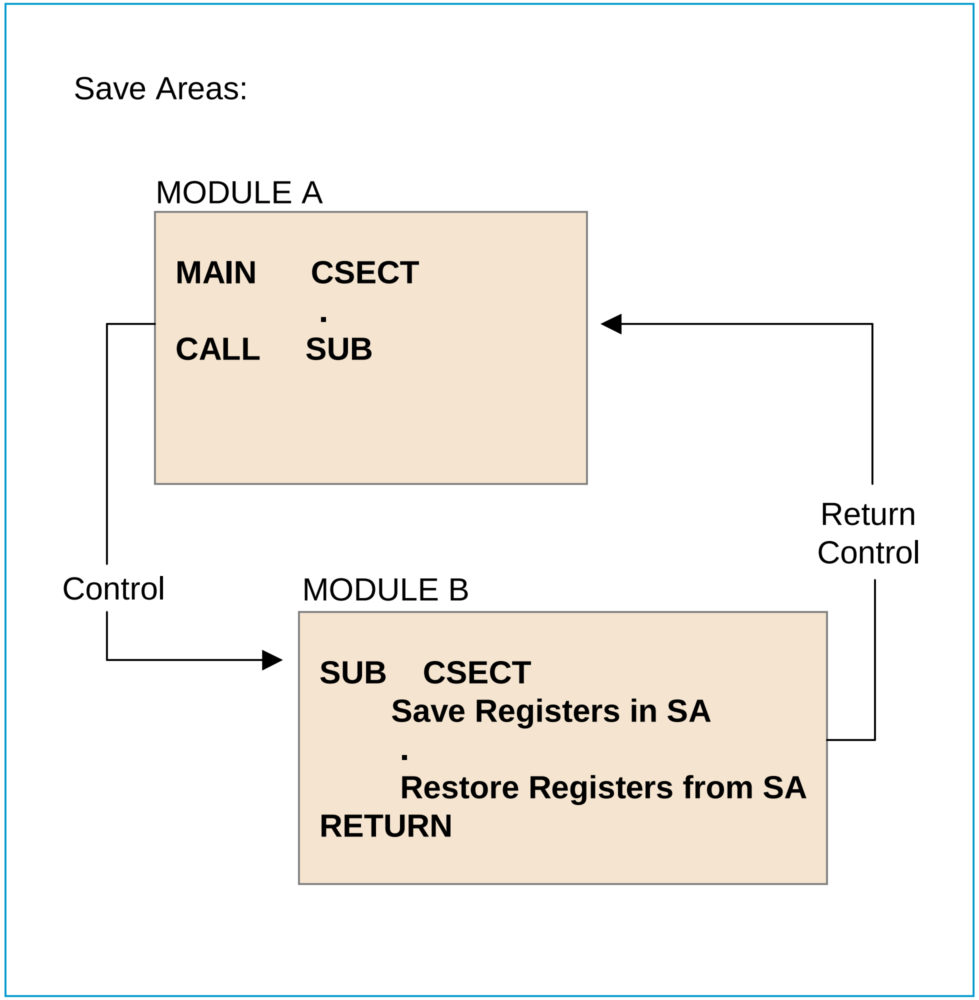
R15- Used for two purposes:

- When control is passed, it contains the address of the instructions in the called module where execution is to begin.
- When control is returned, it contains a return code. This value normally indicates the success of the called program, with 0 representing success.

Save Areas and Linkage

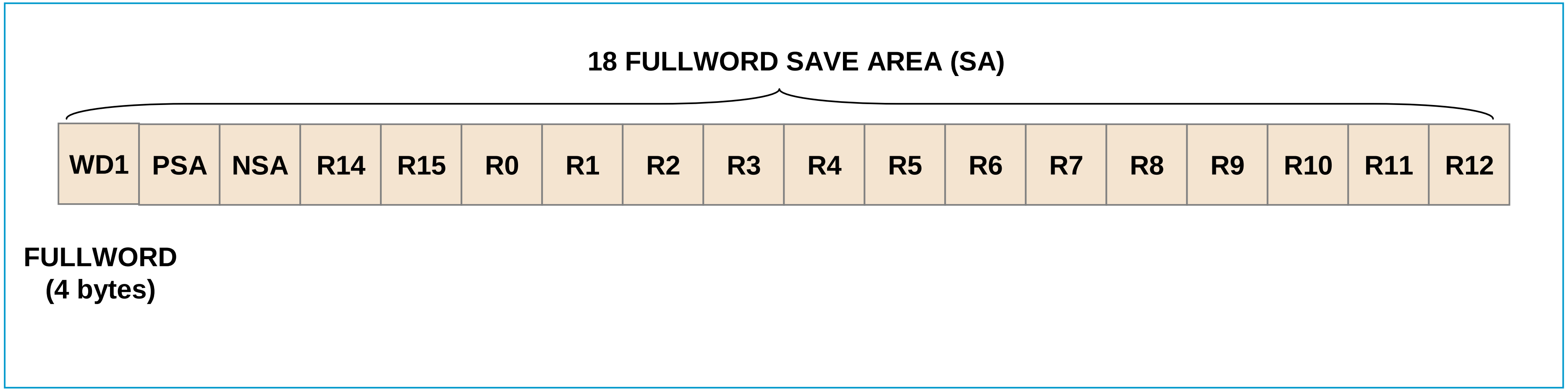
Save areas (SA) are 18 fullword areas that are used to hold register contents when control is passed from module to module.

Each module is responsible for defining a save area for its called programs, in order to save the caller's registers.



Continued…

Save Areas and Linkage (cont'd)



18 FULLWORD SAVE AREA (SA)

WD1 PSA NSA R14 R15 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12

FULLWORD
(4 bytes)

The format of the SA is shown in the here.

PSA is the address of the previous SA.
NSA is the address of the next SA.
R14 holds the contents of R14, which is the return address in the caller.
R15 holds the contents of R15, which is the entry point in the called program.
R0 – R12 contain the contents of registers R0 through R12.

Continued…

Save Areas and Linkage (cont'd)

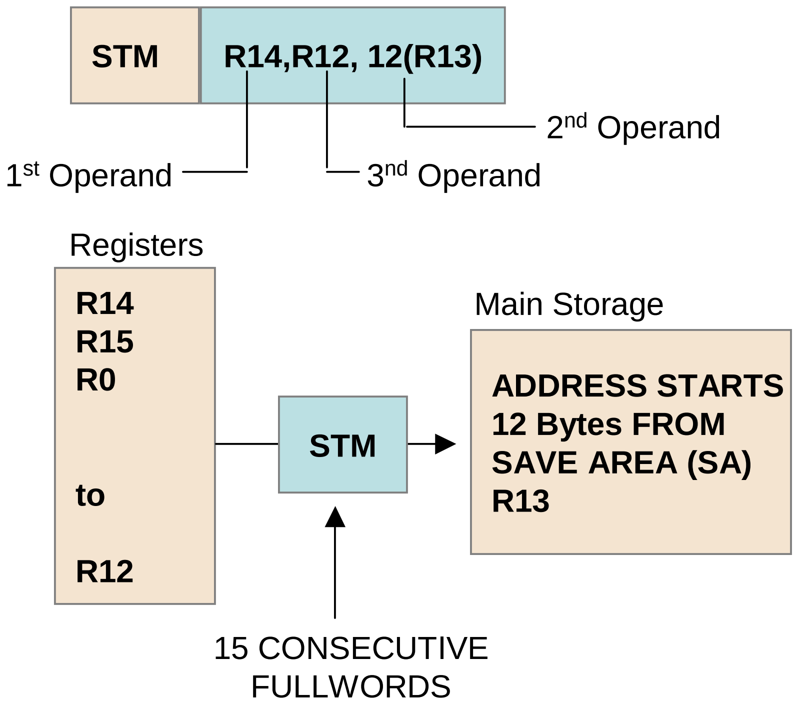
What is a Store Multiple?

The Store Multiple (STM) instruction stores the contents of registers in their correct positions. This is why it is the first instruction one encounters in almost every program.

Example:

STM R14,R12,12(R13)

Store Multiple (STM)
RS Format – Register and Storage

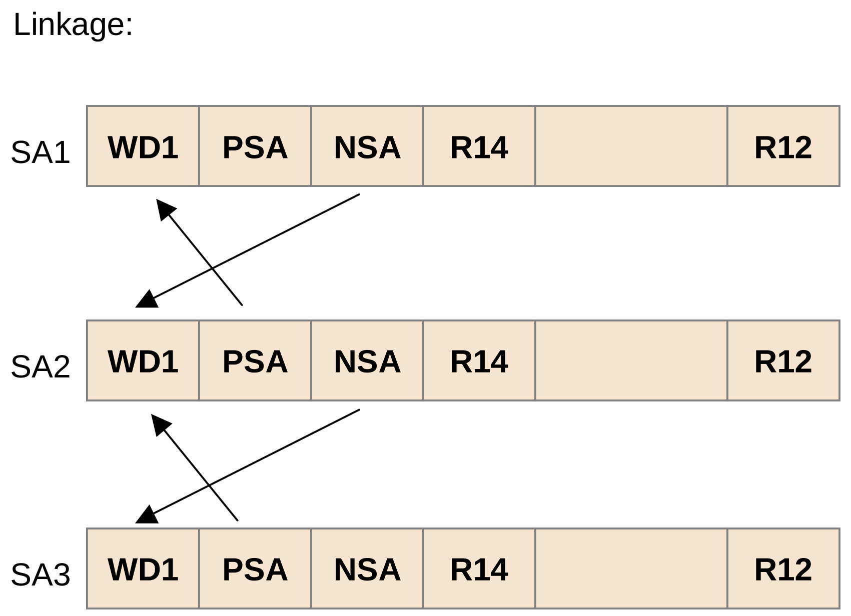


Continued…

Save Areas and Linkage (cont'd)

Each SA points to the Previous Save Area (PSA) and the Next Save Area (NSA).

For this reason, all of the SAs, for active programs, are linked together in a doubly linked list. If a program fails, it is possible to determine the register contents at each point, when control passed from module to module.



Branch and Link

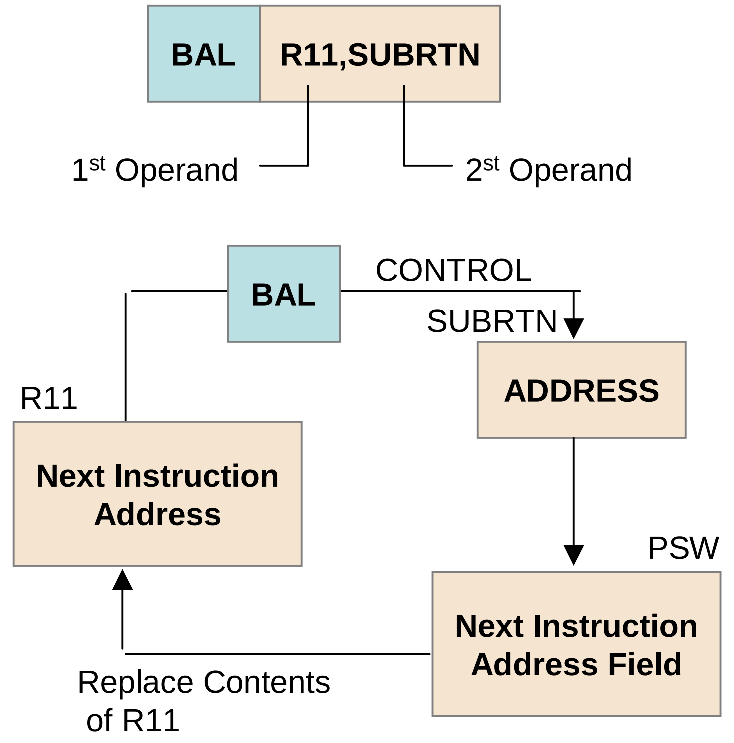
The Assembler Language instructions that implement subroutine linkage are called Branch and Link (BAL).

The next instruction address from the Program Status Word (PSW) is loaded into the GPR specified as the first operand. The address specified as the second operand is placed in the next instruction address field of the PSW.

Example:

BAL R11,SUBRTN

Branch and Link (BAL):
RX Format – Register and Indexed Storage



CC Settings – Condition Code is Unchanged

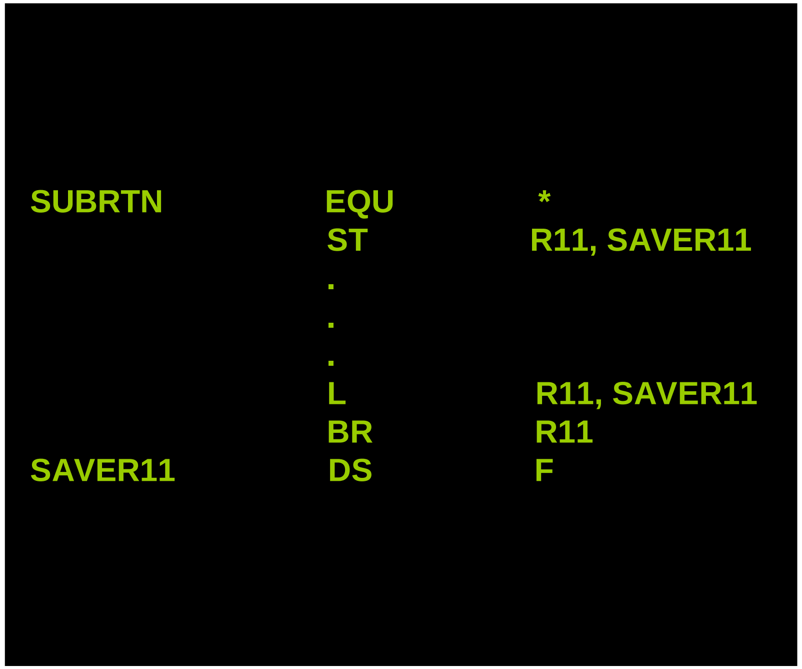
Continued…

Branch and Link (cont'd)

The code at SUBRTN performs its function, and then control can be returned to the caller (the instruction after the BAL), by branching to the address saved in R11.

This can be coded as:

BR R11



```
SUBRTN          EQU        *
                ST         R11, SAVER11
                .
                .
                .
                L          R11, SAVER11
                BR         R11
SAVER11         DS         F
```

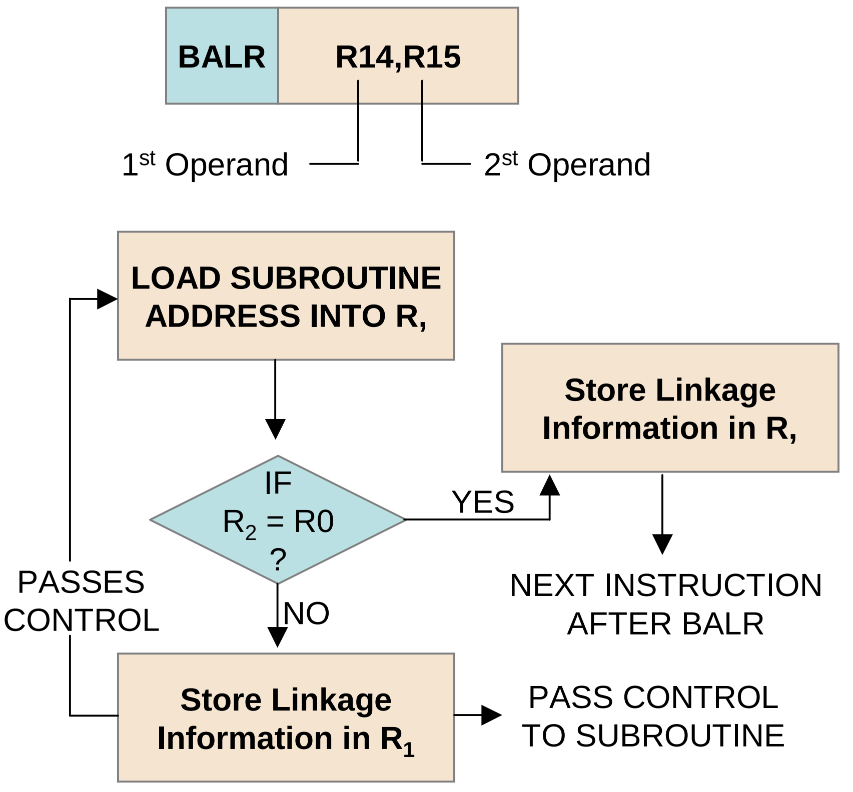
Continued…

Branch and Link (cont'd)

RR Form of BAL instruction

In the RR form of Branch and Link (BALR) instruction, the subroutine address must be loaded into the second operand register before the BALR instruction is executed. If R2 specifies R0, then the linkage information is stored in the first operand, but no branch takes place.

Branch and Link (BALR):
RR Format – Register to Register


BALR
R14,R15
1st Operand
2st Operand
LOAD SUBROUTINE ADDRESS INTO R,
IF R_2 = R0 ?
YES
NO
Store Linkage Information in R,
NEXT INSTRUCTION AFTER BALR
PASSES CONTROL
Store Linkage Information in R_1
PASS CONTROL TO SUBROUTINE

CC Settings – Condition Code is Unchanged

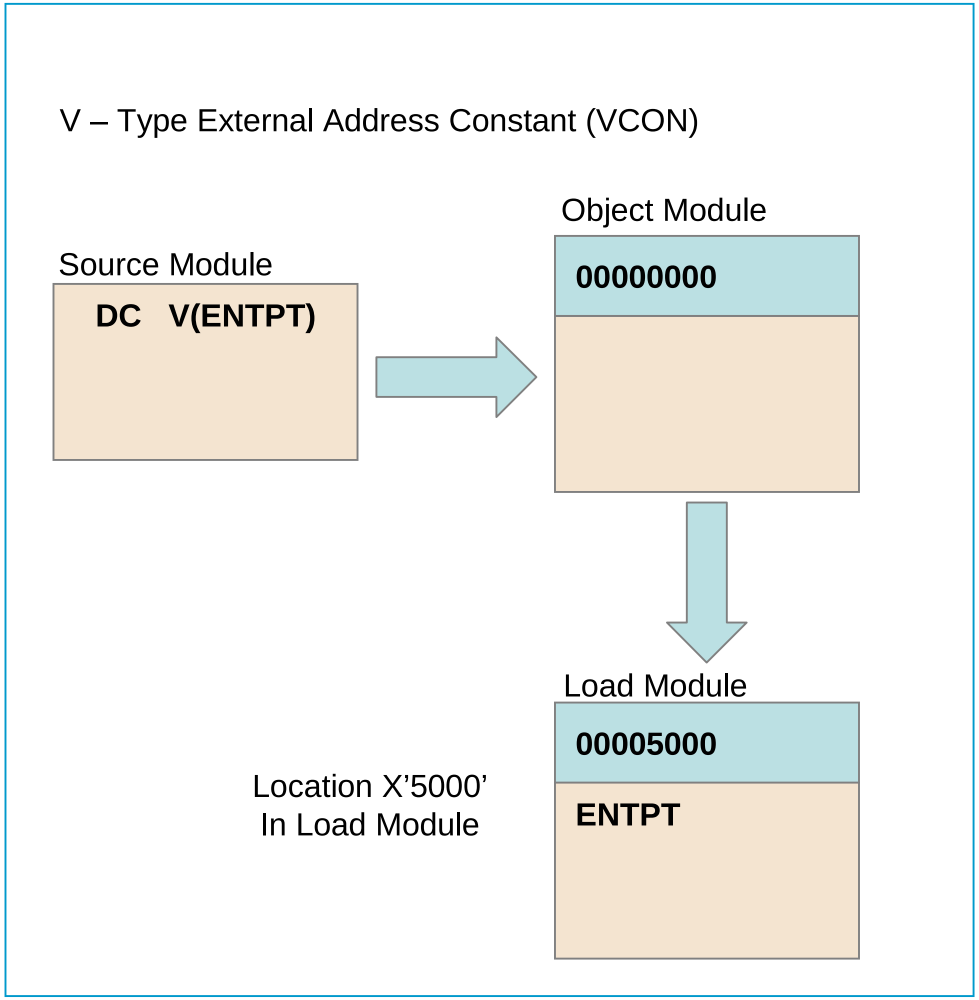
Passing Control

Passing module control involves the use of two GPRs.

In cases where the entry point, to which the control is being passed, is in a separate control section, or in a separately assembled module, a special way is required to reference it.

Assembler Language provides a V-Type external address constant (VCON) to define entry points.

The Linkage Editor adjusts the value of the VCON.



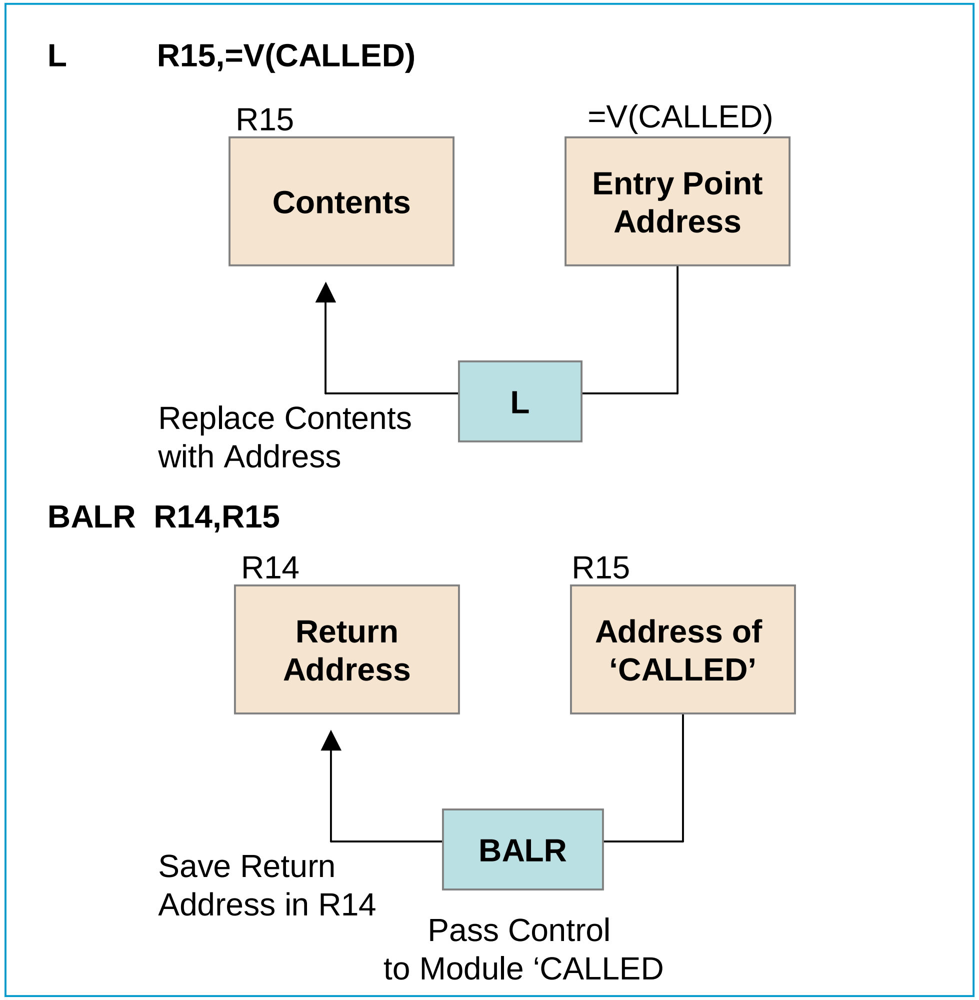
Continued…

Passing Control (cont'd)

To pass control from one module to another, the address constant should be loaded into R15, and then BALR should be used to pass control.

This places the address of the next instruction (the return point) into R14.

The example shows how the control passed to the module CALLED.



L R15,=V(CALLED)

R15

Contents

=V(CALLED)

Entry Point Address

L

Replace Contents with Address

BALR R14,R15

R14

Return Address

R15

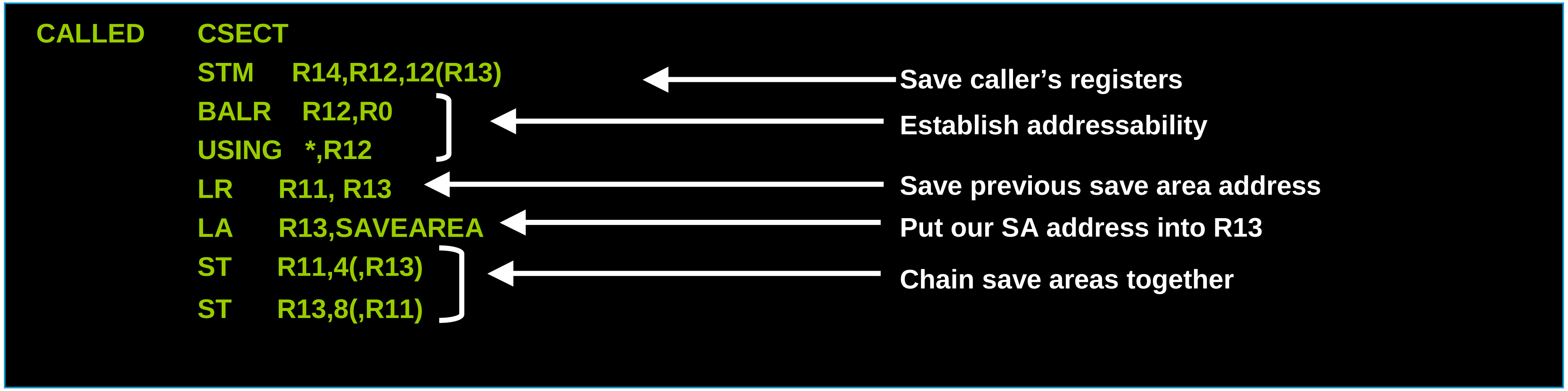
Address of 'CALLED'

BALR

Save Return Address in R14

Pass Control to Module 'CALLED

Receiving Control



```
CALLED   CSECT
         STM    R14,R12,12(R13)
         BALR   R12,R0
         USING  *,R12
         LR     R11, R13
         LA     R13,SAVEAREA
         ST     R11,4(,R13)
         ST     R13,8(,R11)
```

Save caller's registers
Establish addressability
Save previous save area address
Put our SA address into R13
Chain save areas together

The code to receive control in Assembler Language has the following responsibilities:

1. First, the program must save the callers registers
2. Next, addressability is established by designing a base register and loading it with an initial value
3. The program must then set up its own SA and chain the two SAs, the PSA slot and the NSA slot together (so that its SA points to the PSA which then points to the NSA)

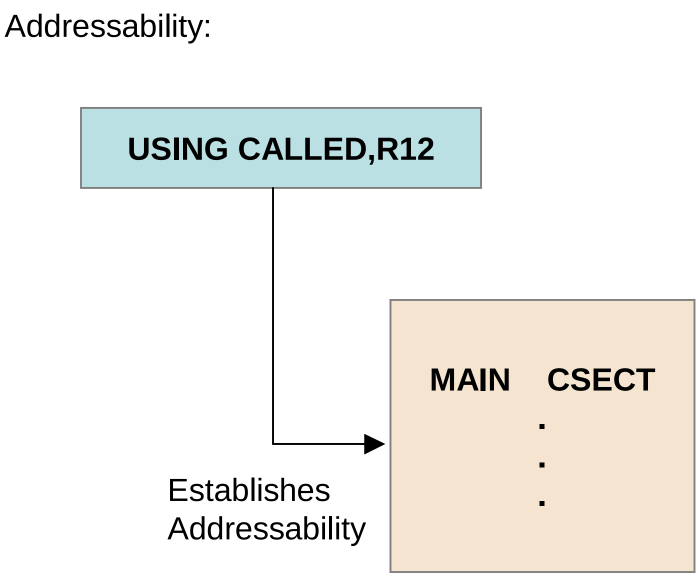
Addressability is established as shown.

Continued…

Receiving Control (cont'd)

Another way to establish addressability uses the entry point address, passed in R15. Instead of BALR and USING, this variant uses the following:

```
LR    R12,R15      Establish
USING CALLED,R12   Addressability
```

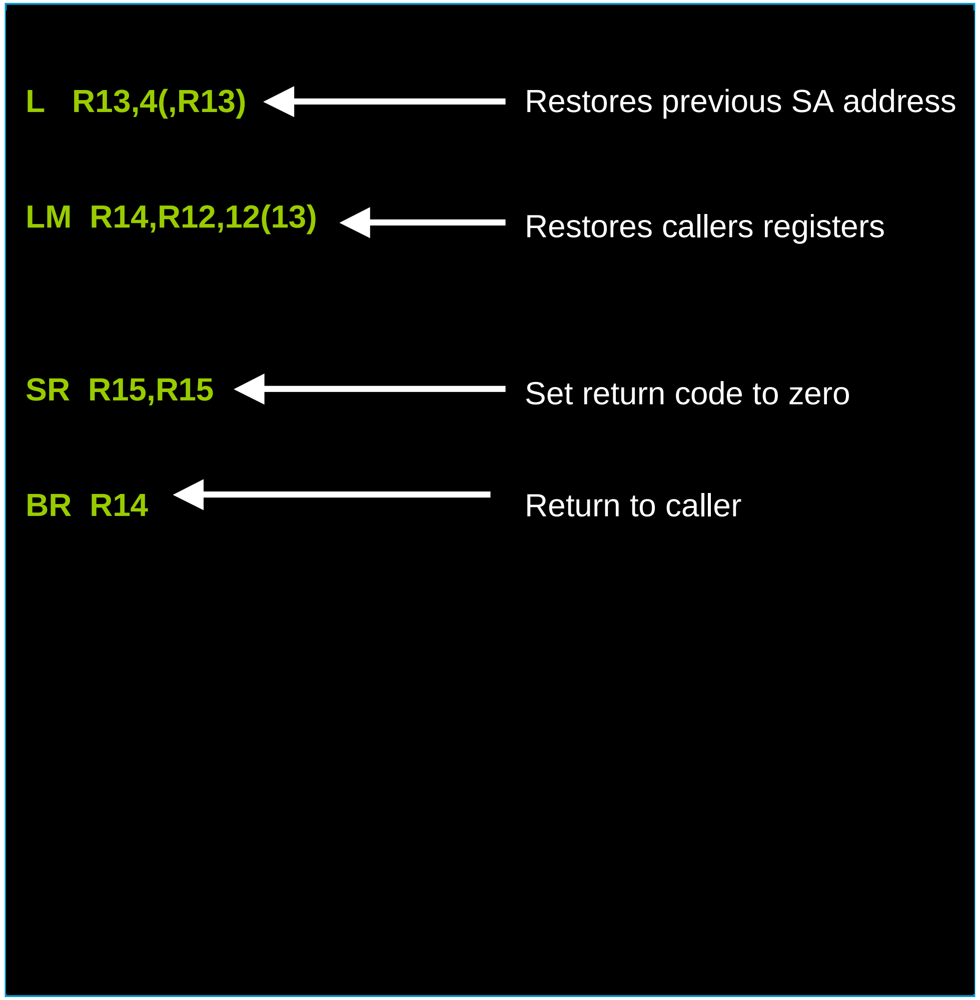


Returning Control

The code to return control back to the caller, must do the following three things:

1. Restore the caller's registers
2. Set the return code
3. Return control through the address that was passed in R14

The code to return control is shown in the graphic. In this example, the return code is set to zero.



```
L   R13,4(,R13)
LM  R14,R12,12(13)
SR  R15,R15
BR  R14
```

Restores previous SA address

Restores callers registers

Set return code to zero

Return to caller

Returning Control (cont'd)

For Return Code Other Than Zero

If a return code is specified to be other than zero, the code will be a little more complex.

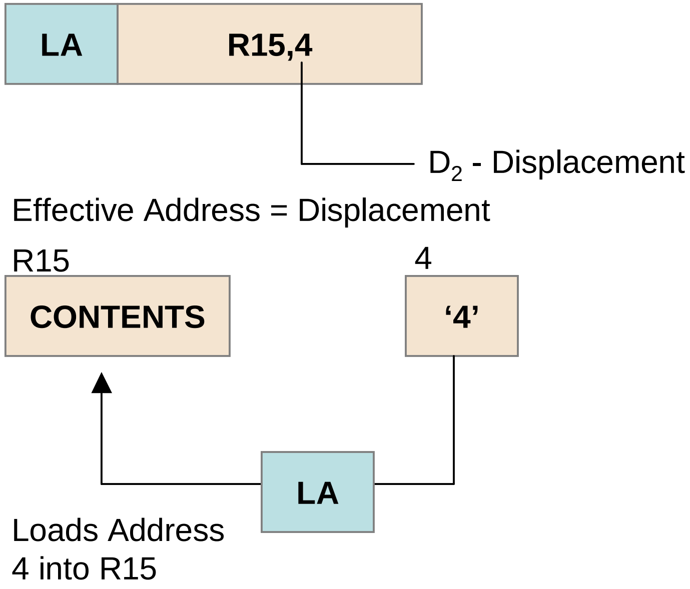
Suppose a return code set to 4 is required, the following code is used:

```
LH R15,=H'4'
```

In order to access the literal, the base register needs to be properly set. However, this instruction has restored the caller's registers, wiping out the proper contents of the base register.

To get around this problem of setting the return code to 4 the following code should be used:

```
LA R15,4
```



Note! This code does not require a base register.

Continued…

Returning Control (cont'd)

For Multiple Possible Return Code

In this case, the code to return control should not be duplicated, so that there is a separate copy for each possible return code.

Instead, return code can be placed in a variable in storage, say a halfword called RC. A single return sequence places this value into R15.

Since you cannot address RC after restoring the caller's registers, set R15 and then restore registers, without overwriting the value in R15.

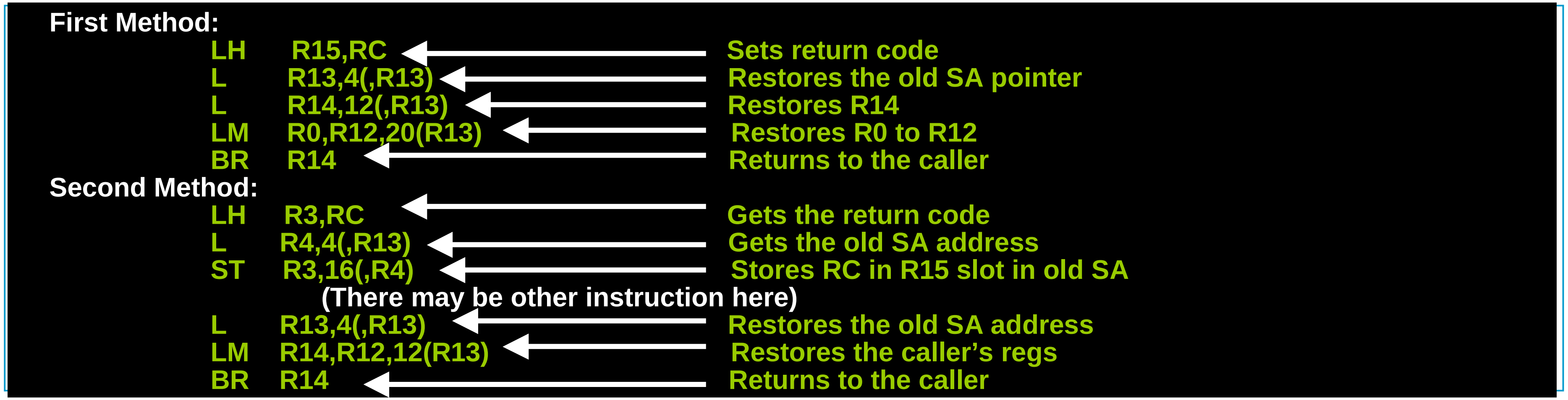
However, this can cause addressability problems.

Addressability Problems:

- **RC Cannot Be Restored After Restore Caller's Registers**
- **R15 Cannot Be Set Prior To Restoring Registers, Without Overwriting Contents of R15**

Continued…

Returning Control: Solution to Addressability Problem



```
First Method:
          LH    R15,RC            Sets return code
          L     R13,4(,R13)       Restores the old SA pointer
          L     R14,12(,R13)      Restores R14
          LM    R0,R12,20(R13)    Restores R0 to R12
          BR    R14               Returns to the caller
Second Method:
                                  Gets the return code
          LH    R3,RC
          L     R4,4(,R13)        Gets the old SA address
          ST    R3,16(,R4)        Stores RC in R15 slot in old SA
                (There may be other instruction here)
          L     R13,4(,R13)       Restores the old SA address
          LM    R14,R12,12(R13)   Restores the caller's regs
          BR    R14               Returns to the caller
```

There are two possible methods to solve addressability problem. The methods are explained here:

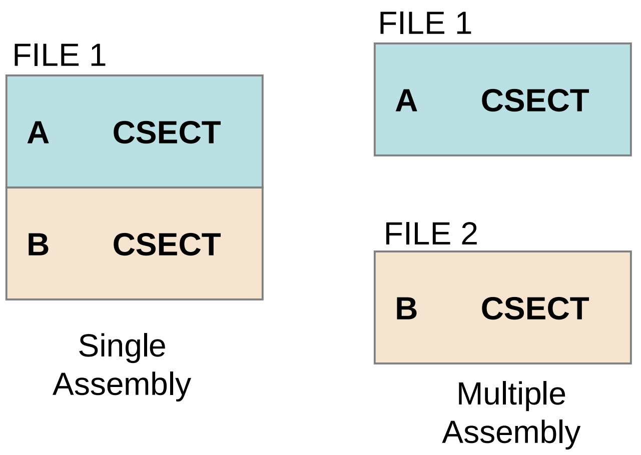
- In the first method, the value can be set into R15, and then the registers can be restored, except for R15
- The other method is to put the desired return code value not into R15, but into R15 slot in the caller's save area. It will then be restored when the Load Multiple is completed prior to returning.

Single and Multiple Assemblies

While creating a multi-modular Assembler Language program, modules can be assembled in a single assembly or assembled separately.

Assembling the modules together provides the advantage of only having a single source file to manage.

The main disadvantage is that names must be unique within an assembly.



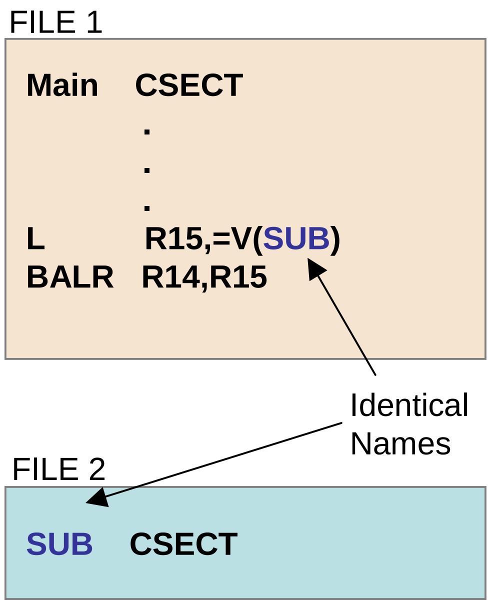
Continued…

Single and Multiple Assemblies (cont'd)

When the number of modules exceeds 3 or 4, it is better to use separate source files.

The linkage editor combines individual object modules into a single, executable module.

It does this by matching external references (V-type adcons in your program), with external CSECT names in your program.



FILE 1

```
Main   CSECT
        .
        .
        .
L       R15,=V(SUB)
BALR   R14,R15
```

Identical Names

FILE 2

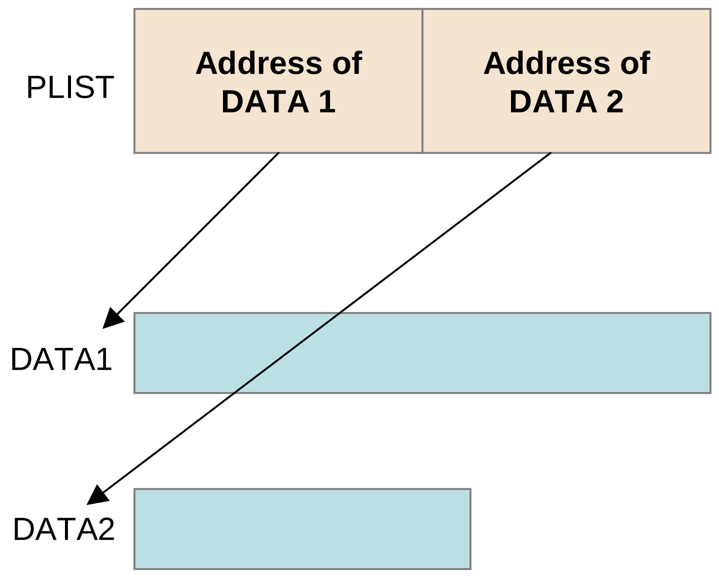
```
SUB   CSECT
```

Parameter Lists

The mechanism used for passing data between modules is the parameter list. A parameter can be defined as a list of the addresses of the data being passed from caller to called module.

Each entry in the parameter list is a fullword, containing the address of the corresponding data. Regardless of whether the actual data is a single byte in length, or many thousands of bytes, it is the address of the data that is passed, and this fits in a fullword.

2 Entry Parameter List:



Continued…

Parameter Lists (cont'd)

Fixed-Length Parameter List

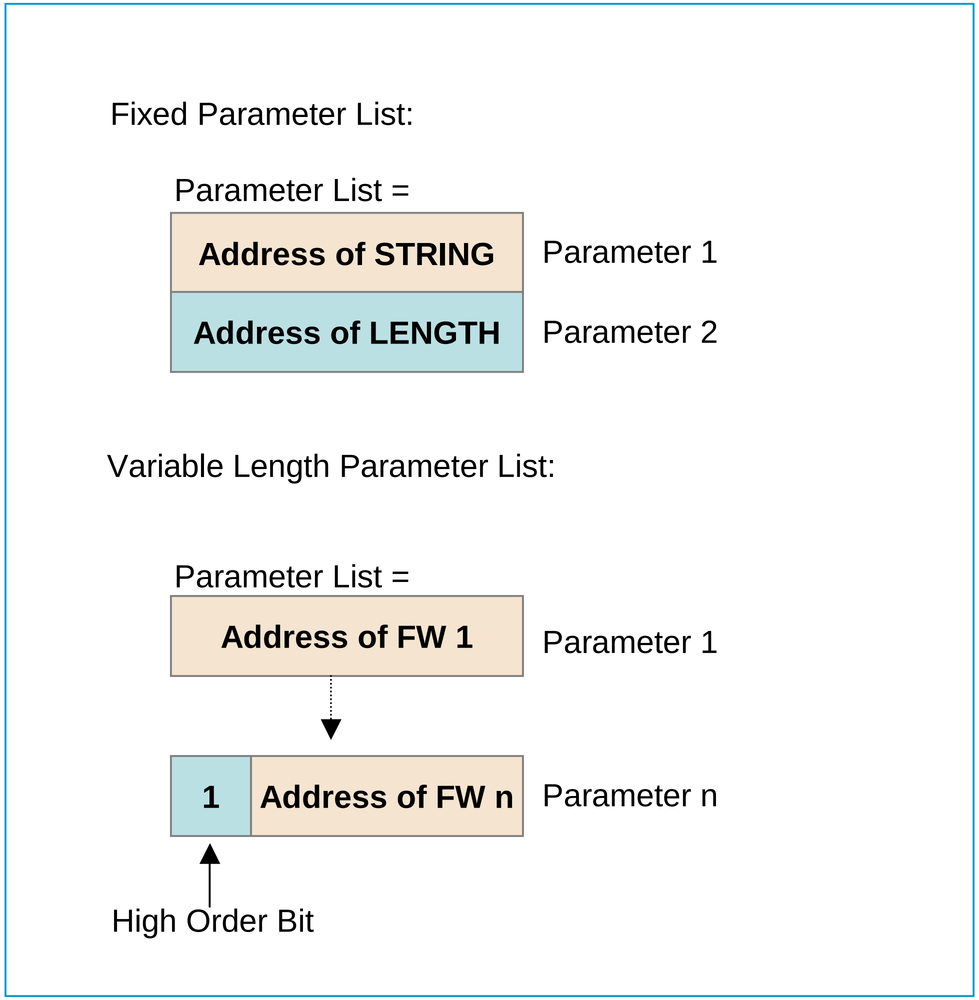
In some cases, a module is designed to accept a fixed number of parameters.

Consider a module to determine the length of a string. It might be designed to use two parameters: one to contain the string, the other to hold the length. The parameter list to this module is fixed in length with two entries.

Variable Length Parameter List

In other cases modules are designed to accept any number of parameters.

Consider a module to calculate the maximum value of a group of fullwords. It might receive 2, or 10, or 100 fullwords as parameters. The parameter list for this module is variable in length.



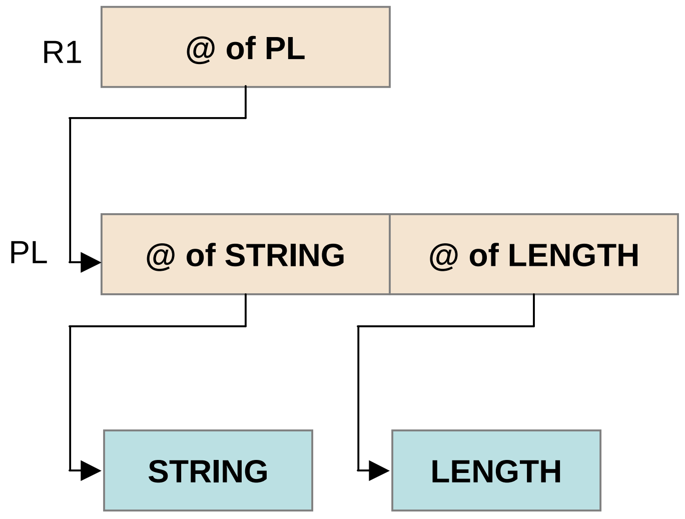
Continued…

Parameter Lists (cont'd)

When control is passed from one module to another, the address of the parameter list is placed in R1.

So, R1 contains the address of the parameter list, which itself is a list of addresses.

Passing Control:

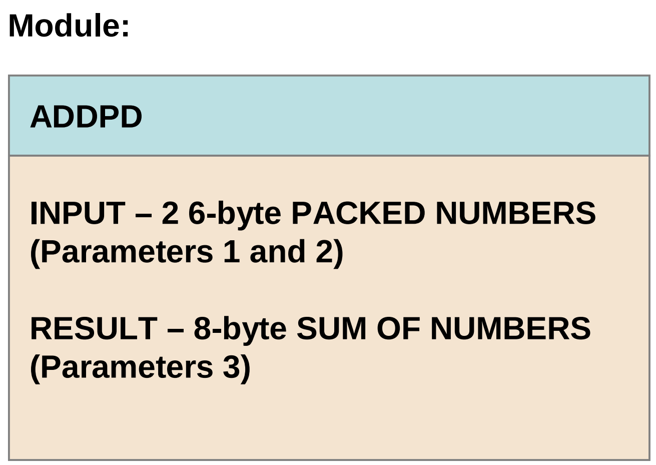


Fixed Length Parameter Lists

Consider a case of adding two packed decimal numbers, and returning the sum.

A separate module, called ADDPD, is used here to perform this simple operation.

This module receives three parameters. The first two parameters are the two 6-byte packed decimal numbers to be added together, and the third parameter is an eight byte result field.



Module:

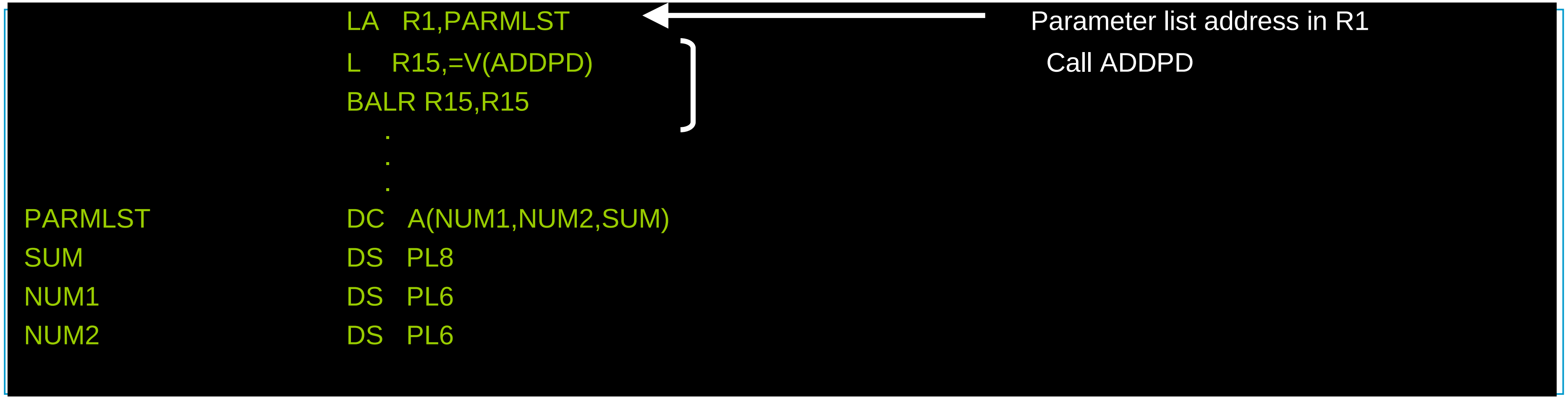
ADDPD

INPUT – 2 6-byte PACKED NUMBERS (Parameters 1 and 2)

RESULT – 8-byte SUM OF NUMBERS (Parameters 3)

Continued…

Fixed Length Parameter Lists (cont’d)



```
                    LA   R1,PARMLST
                    L    R15,=V(ADDPD)
                    BALR R15,R15
                      .
                      .
                      .
PARMLST             DC   A(NUM1,NUM2,SUM)
SUM                 DS   PL8
NUM1                DS   PL6
NUM2                DS   PL6
```

Parameter list address in R1

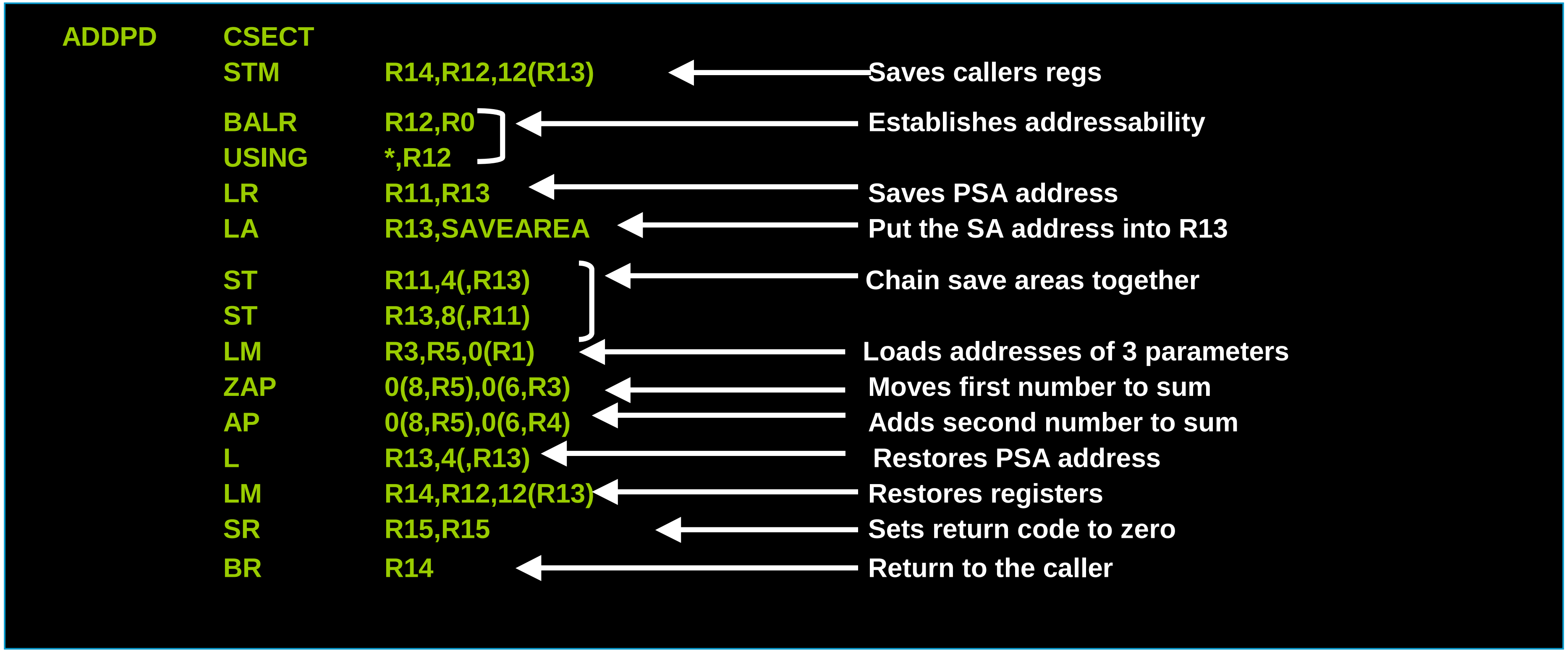
Call ADDPD

This example shows the code necessary to call ADDPD module.

In addition to the code for passing control, here, the parameter list has been set up and its address has been placed in R1.

Continued…

Fixed Length Parameter Lists (cont’d)

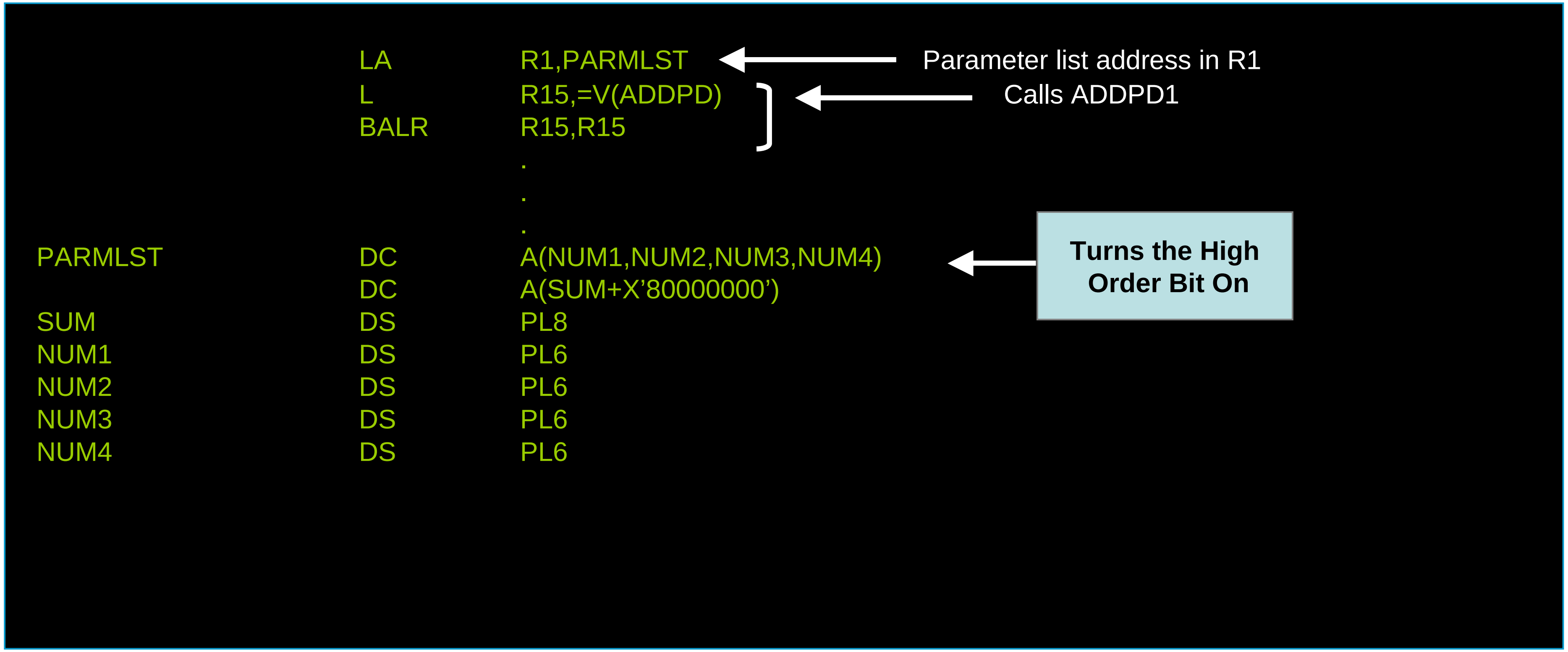


```
ADDPD    CSECT
         STM      R14,R12,12(R13)         Saves callers regs
         BALR     R12,R0                  Establishes addressability
         USING    *,R12
         LR       R11,R13                 Saves PSA address
         LA       R13,SAVEAREA            Put the SA address into R13
         ST       R11,4(,R13)             Chain save areas together
         ST       R13,8(,R11)
         LM       R3,R5,0(R1)             Loads addresses of 3 parameters
         ZAP      0(8,R5),0(6,R3)         Moves first number to sum
         AP       0(8,R5),0(6,R4)         Adds second number to sum
         L        R13,4(,R13)             Restores PSA address
         LM       R14,R12,12(R13)         Restores registers
         SR       R15,R15                 Sets return code to zero
         BR       R14                     Return to the caller
```

This example shows how to code ADDPD.

Since ADDPD is receiving the addresses of the parameters, it must load them into registers, and then use explicit base displacement addresses, with length, to access them.

Variable Length Parameter Lists



```
                LA      R1,PARMLST
                L       R15,=V(ADDPD)
                BALR    R15,R15
                        .
                        .
                        .
PARMLST         DC      A(NUM1,NUM2,NUM3,NUM4)
                DC      A(SUM+X’80000000’)
SUM             DS      PL8
NUM1            DS      PL6
NUM2            DS      PL6
NUM3            DS      PL6
NUM4            DS      PL6
```

Parameter list address in R1

Calls ADDPD1

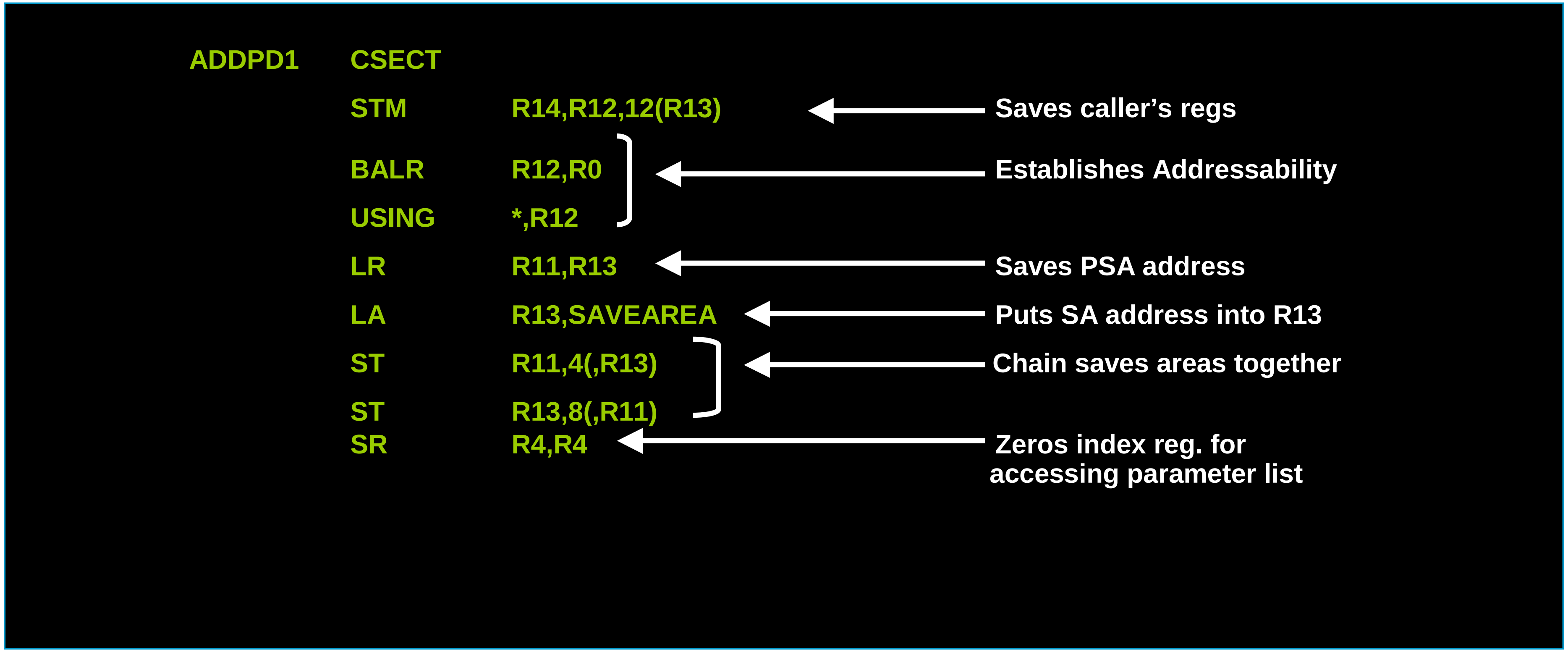
Turns the High Order Bit On

A module called ADDPD1 has been shown in the example. This illustrates the variable-length parameter list.

ADDPD1 sums a set of 6-byte packed fields into an 8-sum field, provided as the last parameter. The code to call ADDPD1 passes it 4 numbers to add, plus the result field, but the code of ADDPD1 will work with any number of parameters from 1 up.

Continued…

Variable Length Parameter Lists (cont’d)



```
ADDPD1   CSECT
         STM      R14,R12,12(R13)
         BALR     R12,R0
         USING    *,R12
         LR       R11,R13
         LA       R13,SAVEAREA
         ST       R11,4(,R13)
         ST       R13,8(,R11)
         SR       R4,R4
```

Saves caller’s regs

Establishes Addressability

Saves PSA address

Puts SA address into R13

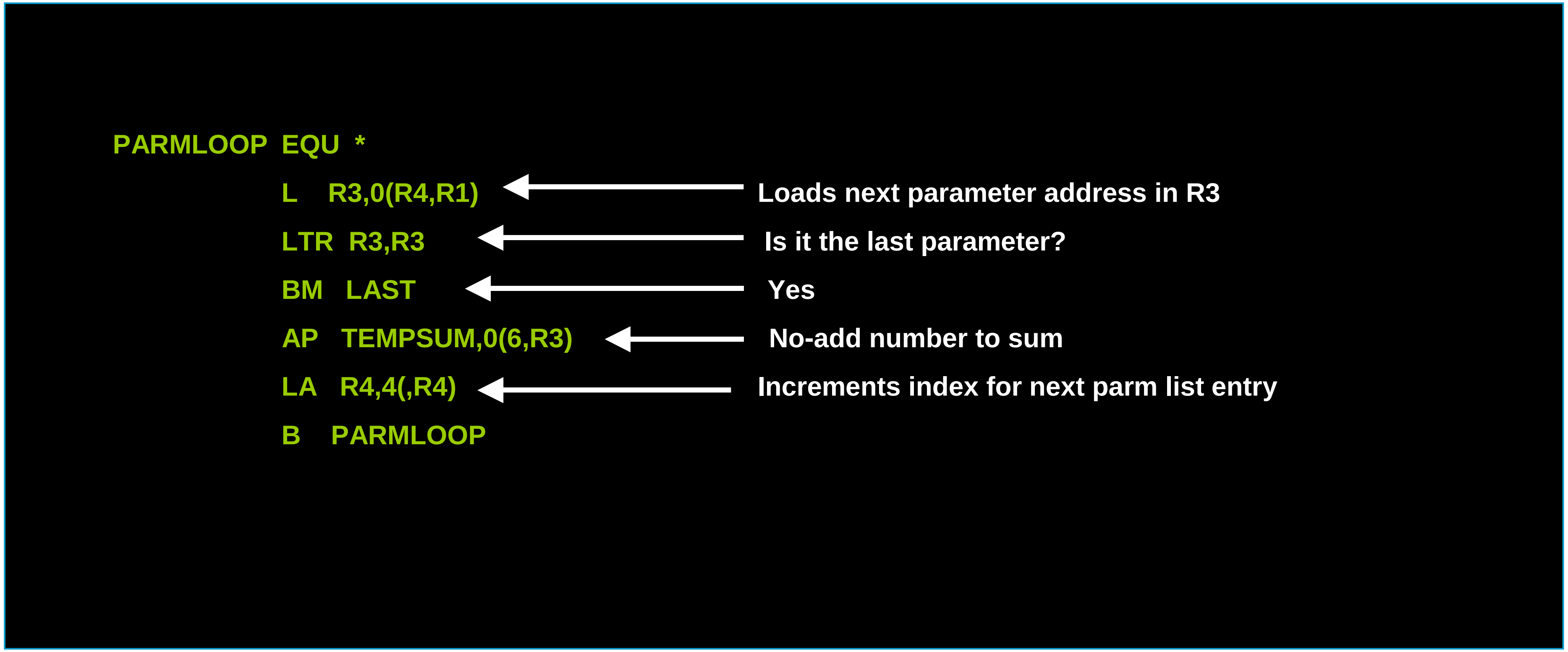
Chain saves areas together

Zeros index reg. for accessing parameter list

This example shows the code for ADDPD1.

Continued…

Variable Length Parameter Lists (cont’d)



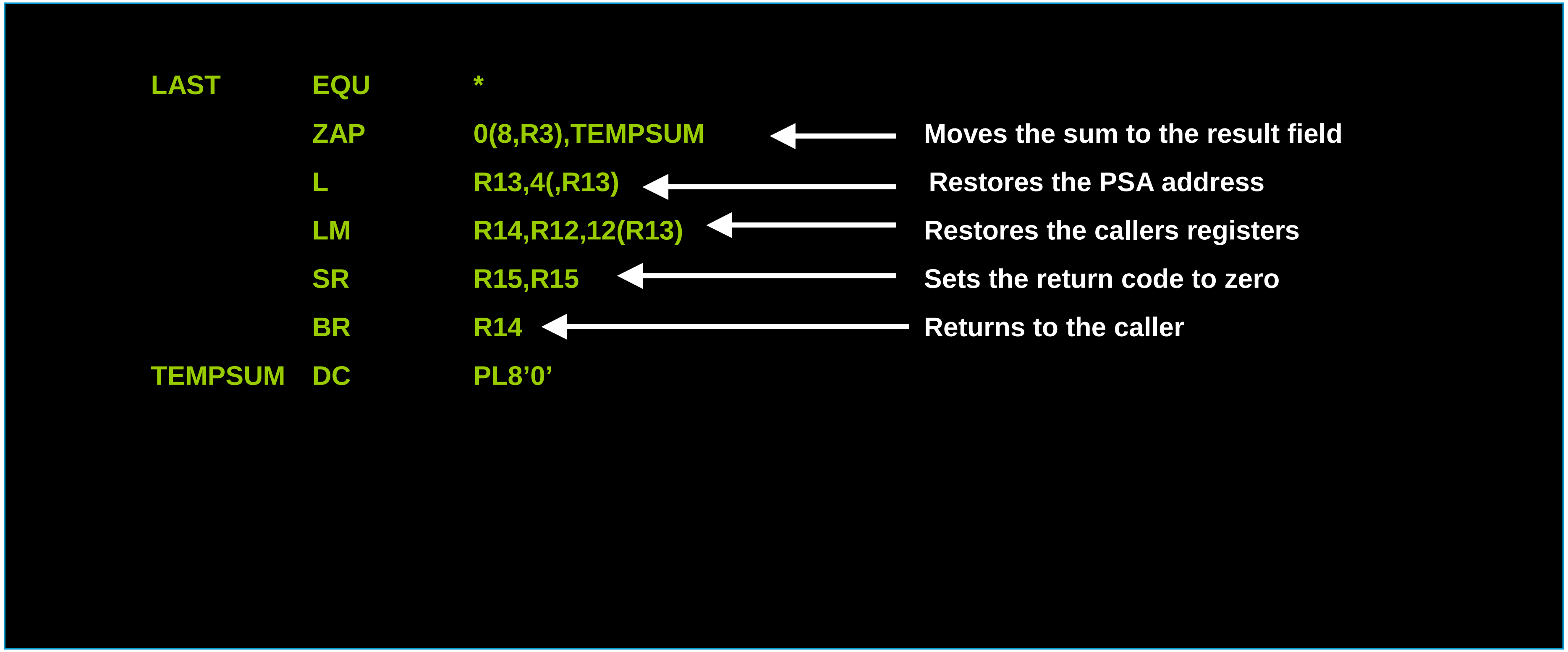
```
PARMLOOP  EQU  *
          L    R3,0(R4,R1)          Loads next parameter address in R3
          LTR  R3,R3                Is it the last parameter?
          BM   LAST                 Yes
          AP   TEMPSUM,0(6,R3)      No-add number to sum
          LA   R4,4(,R4)            Increments index for next parm list entry
          B    PARMLOOP
```

Next, the last parameter needs to be detected by testing each address to see if it is negative. If a fullword is negative, the high order bit is one.

This example shows the code information.

Continued…

Variable Length Parameter Lists (cont'd)



```
LAST      EQU     *
          ZAP     0(8,R3),TEMPSUM
          L       R13,4(,R13)
          LM      R14,R12,12(R13)
          SR      R15,R15
          BR      R14
TEMPSUM   DC      PL8'0'
```

Moves the sum to the result field

Restores the PSA address

Restores the callers registers

Sets the return code to zero

Returns to the caller

The address of the result field remains unknown till the end of the parameter list. In this case, the programmer must sum the numbers into a temporary sum field, and then move that field to the result field, after it is found.

Dummy Sections

Sometimes processed parameters are complex data structures, rather than simple variables.

Consider the case of a customer record, consisting of name, two lines of address, city, state, and zip code as a parameter. The example, here, shows one such record.

In the module where this record was defined, the whole record can be referred to, or any of its fields. It is because, the technique of defining the whole record with a zero duplication factor has been used.

Record:

```
CREC      DS    OCL96
CNAME     DS    CL30
CADDR1    DS    CL20
CADDR2    DS    CL20
CCITY     DS    CL15
CSTATE    DS    CL2
CZIP      DS    CL9
```

Continued…

Dummy Sections (cont'd)

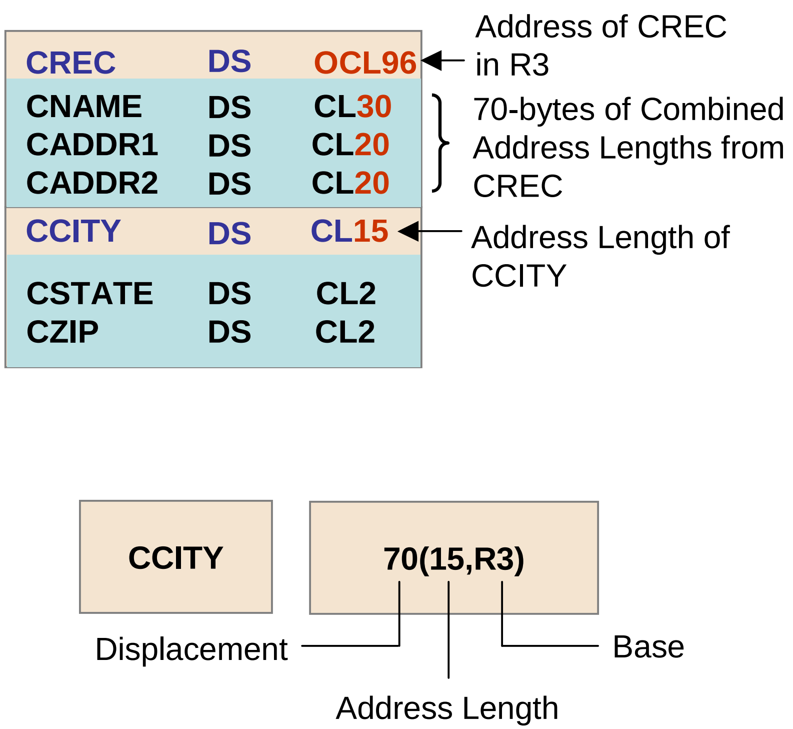
Consider a case where this customer record is passed to another module.

The problem that occurs here is that, in the called module, though there is a need to access the data, it is not defined here. The definitions is in the calling module.

In this case, the need is to specify the structure of data, without actually setting up any storage for it. This is because the storage already exists in another module.

It is done with dummy section (DSECT).

Called Module:



```
CREC      DS    OCL96
CNAME     DS    CL30
CADDR1    DS    CL20
CADDR2    DS    CL20
CCITY     DS    CL15
CSTATE    DS    CL2
CZIP      DS    CL2
```

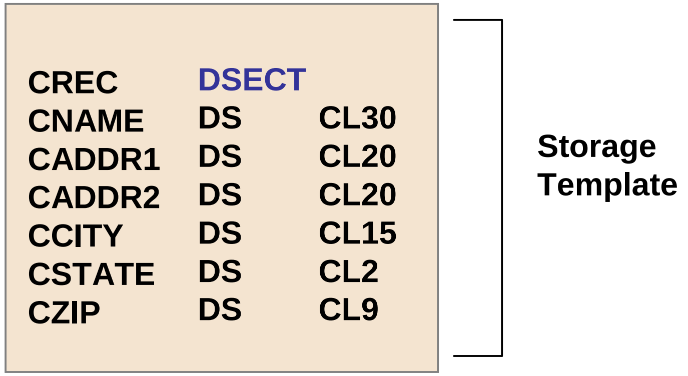
Continued…

Dummy Sections (cont'd)

A DSECT allows the Assembler Language programmer to define a storage template, a structure that defines names and relationships, without allocating any storage.

A DSECT to represent the fields in the customer record can be defined in the separately assembled called module, as shown in the example.

Called Module:



```
CREC      DSECT
CNAME     DS      CL30
CADDR1    DS      CL20
CADDR2    DS      CL20
CCITY     DS      CL15
CSTATE    DS      CL2
CZIP      DS      CL9
```

Continued…

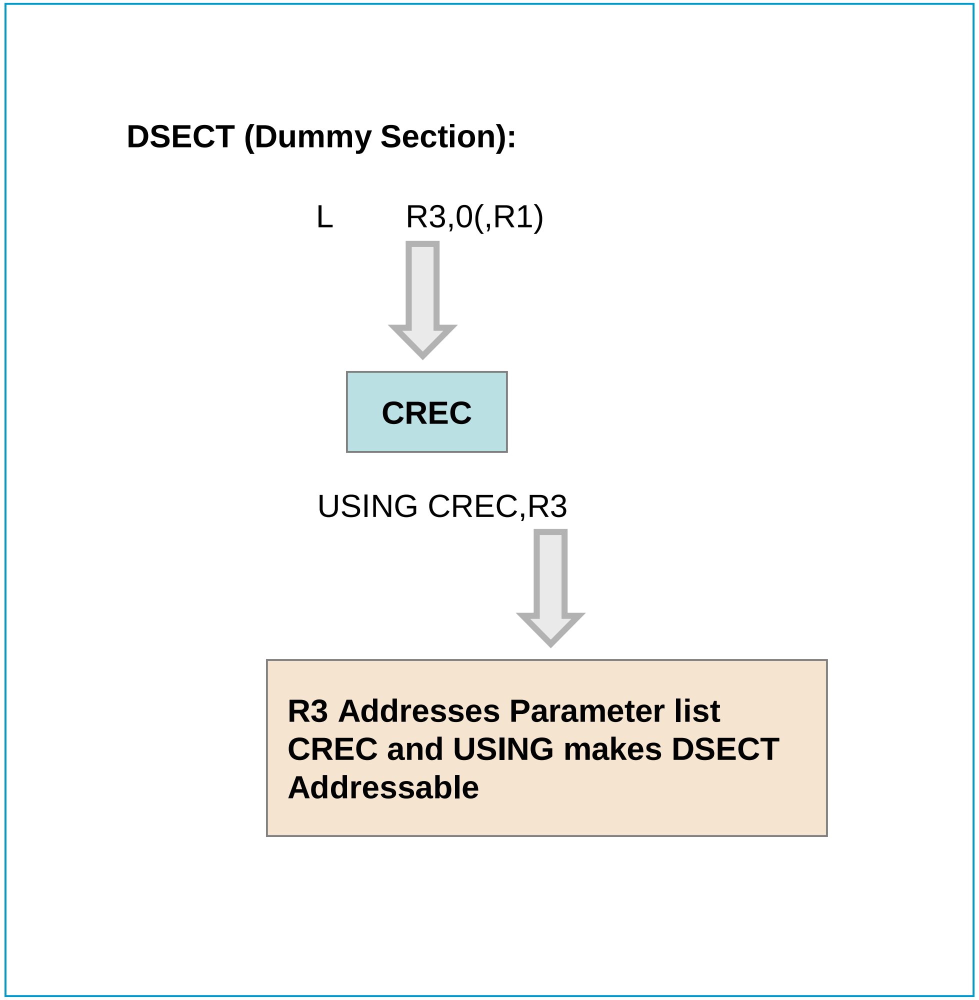
Dummy Sections (cont'd)

How to use DSECT

Given the example of customer record, before referring to CREC or any of its fields, it needs to be made addressable. This is done with a USING instruction.

If the address of the actual storage of CREC is passed in a parameter list, the following code can be used to make it addressable:

```
L     R3,0(,R1)  R3       points to CREC
USING CREC, R3           R3 makes the CREC
                            addressable
```



DSECT (Dummy Section):

```
L      R3,0(,R1)
```

```
USING CREC,R3
```

R3 Addresses Parameter list CREC and USING makes DSECT Addressable

Continued…

Dummy Sections (cont'd)

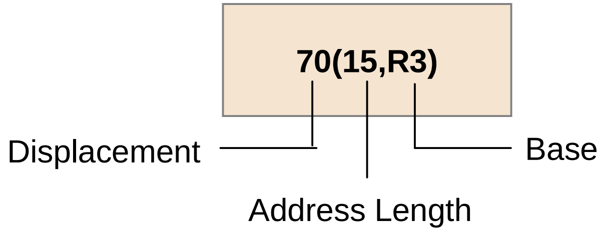
At this point, fields of CREC can be referred to by name. The Assembler translates a reference to CCITY as 70(15,R3).

This is because CCITY is 70 bytes beyond CREC, and Assembler has been told that R3 contains the base address of CREC.

The same Machine Language code can be generated without the DSECT. The use of DSECT makes the code clearer, since it uses a symbol, rather than a base displacement address.

The use of DSECTs should always be considered while dealing with structured data defined in another module, and passed as a parameter.

Fields:



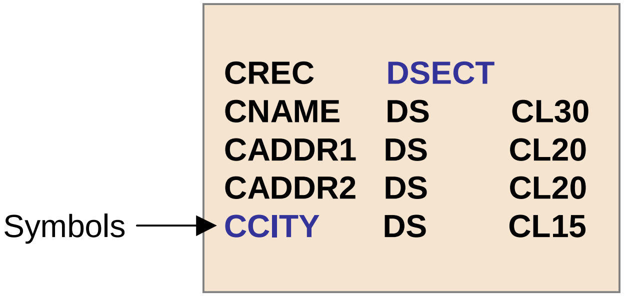
70(15,R3)

Displacement

Base

Address Length

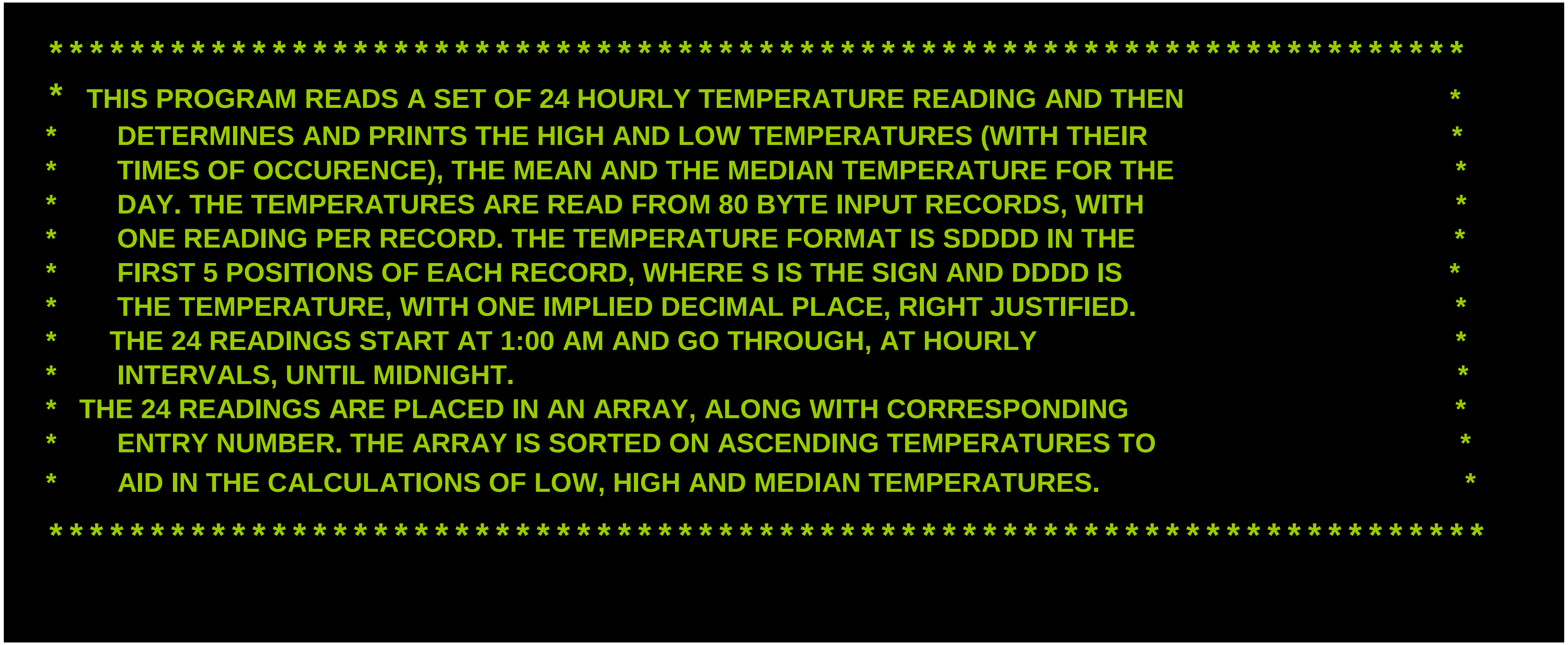
Symbols:



Symbols

```
CREC      DSECT
CNAME     DS       CL30
CADDR1    DS       CL20
CADDR2    DS       CL20
CCITY     DS       CL15
```

The Main Module

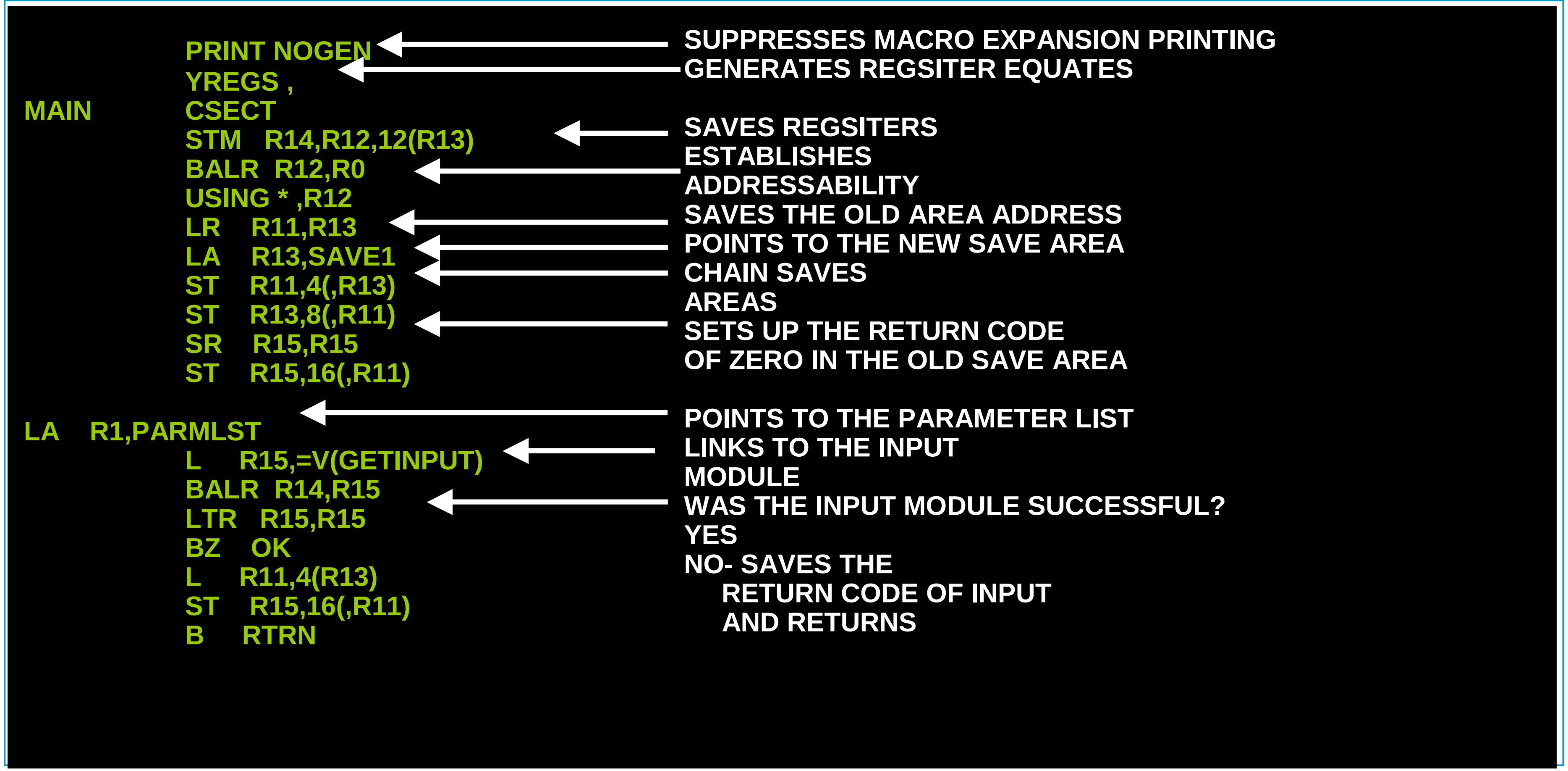


```
**********************************************************************
*  THIS PROGRAM READS A SET OF 24 HOURLY TEMPERATURE READING AND THEN      *
*      DETERMINES AND PRINTS THE HIGH AND LOW TEMPERATURES (WITH THEIR     *
*      TIMES OF OCCURENCE), THE MEAN AND THE MEDIAN TEMPERATURE FOR THE    *
*      DAY. THE TEMPERATURES ARE READ FROM 80 BYTE INPUT RECORDS, WITH     *
*      ONE READING PER RECORD. THE TEMPERATURE FORMAT IS SDDDD IN THE      *
*      FIRST 5 POSITIONS OF EACH RECORD, WHERE S IS THE SIGN AND DDDD IS   *
*      THE TEMPERATURE, WITH ONE IMPLIED DECIMAL PLACE, RIGHT JUSTIFIED.   *
*     THE 24 READINGS START AT 1:00 AM AND GO THROUGH, AT HOURLY           *
*      INTERVALS, UNTIL MIDNIGHT.                                          *
*  THE 24 READINGS ARE PLACED IN AN ARRAY, ALONG WITH CORRESPONDING        *
*      ENTRY NUMBER. THE ARRAY IS SORTED ON ASCENDING TEMPERATURES TO      *
*      AID IN THE CALCULATIONS OF LOW, HIGH AND MEDIAN TEMPERATURES.       *
***********************************************************************
```

Each module should start with a block of comments explaining its purpose. The header comments for the main module are shown in the example.

Continued…

The Main Module (cont'd)



```
            PRINT NOGEN
            YREGS ,
MAIN        CSECT
            STM   R14,R12,12(R13)
            BALR  R12,R0
            USING * ,R12
            LR    R11,R13
            LA    R13,SAVE1
            ST    R11,4(,R13)
            ST    R13,8(,R11)
            SR    R15,R15
            ST    R15,16(,R11)

LA    R1,PARMLST
            L     R15,=V(GETINPUT)
            BALR  R14,R15
            LTR   R15,R15
            BZ    OK
            L     R11,4(R13)
            ST    R15,16(,R11)
            B     RTRN
```

SUPPRESSES MACRO EXPANSION PRINTING
GENERATES REGSITER EQUATES

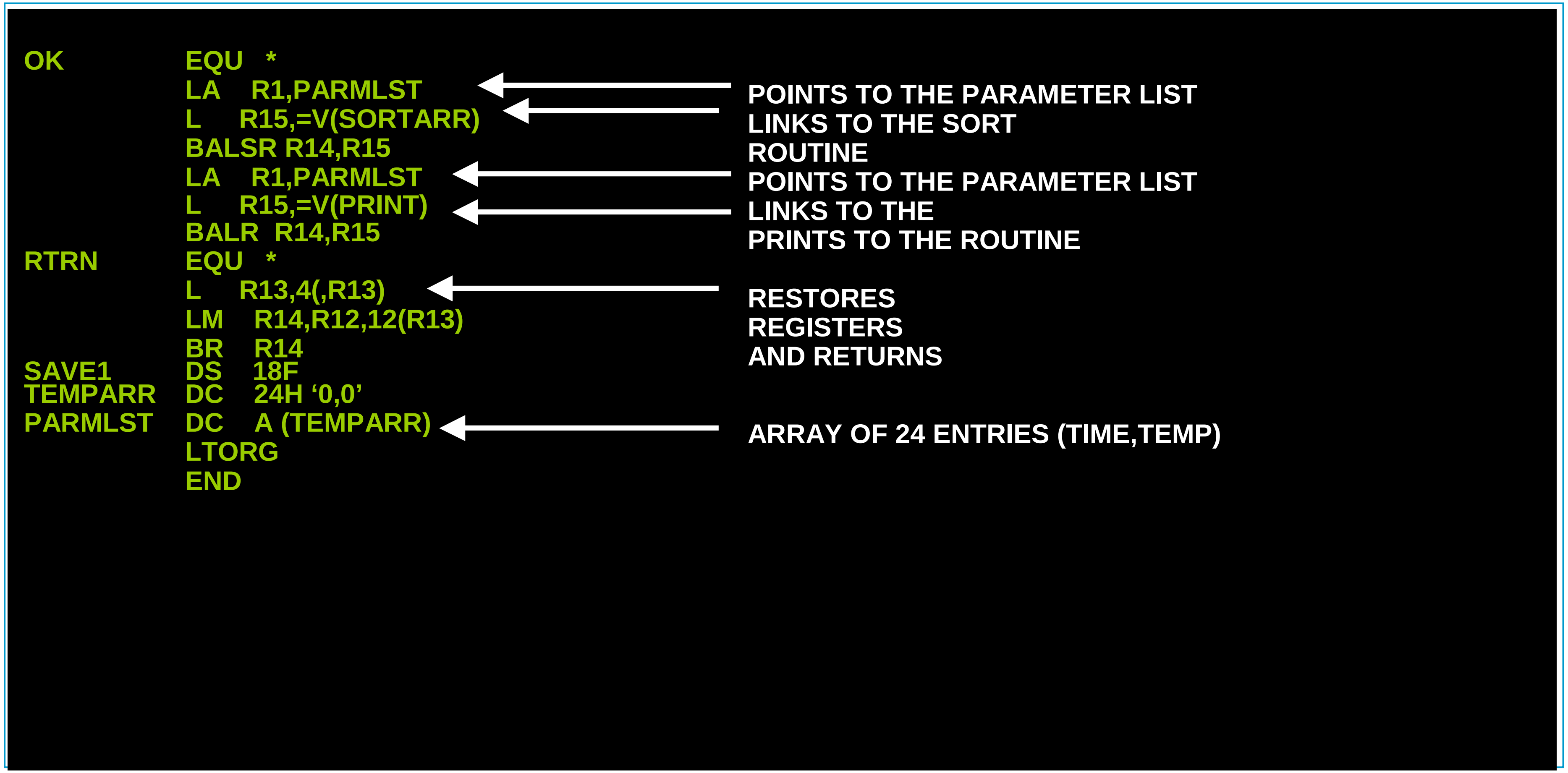
SAVES REGSITERS
ESTABLISHES
ADDRESSABILITY
SAVES THE OLD AREA ADDRESS
POINTS TO THE NEW SAVE AREA
CHAIN SAVES
AREAS
SETS UP THE RETURN CODE
OF ZERO IN THE OLD SAVE AREA

POINTS TO THE PARAMETER LIST
LINKS TO THE INPUT
MODULE
WAS THE INPUT MODULE SUCCESSFUL?
YES
NO- SAVES THE
  RETURN CODE OF INPUT
  AND RETURNS

This example shows a part of the coding for main module.

Continued…

The Main Module (cont’d)



```
OK       EQU   *
         LA    R1,PARMLST
         L     R15,=V(SORTARR)
         BALSR R14,R15
         LA    R1,PARMLST
         L     R15,=V(PRINT)
         BALR  R14,R15
RTRN     EQU   *
         L     R13,4(,R13)
         LM    R14,R12,12(R13)
         BR    R14
SAVE1    DS    18F
TEMPARR  DC    24H ‘0,0’
PARMLST  DC    A (TEMPARR)
         LTORG
         END
```

POINTS TO THE PARAMETER LIST
LINKS TO THE SORT
ROUTINE
POINTS TO THE PARAMETER LIST
LINKS TO THE
PRINTS TO THE ROUTINE

RESTORES
REGISTERS
AND RETURNS

ARRAY OF 24 ENTRIES (TIME,TEMP)

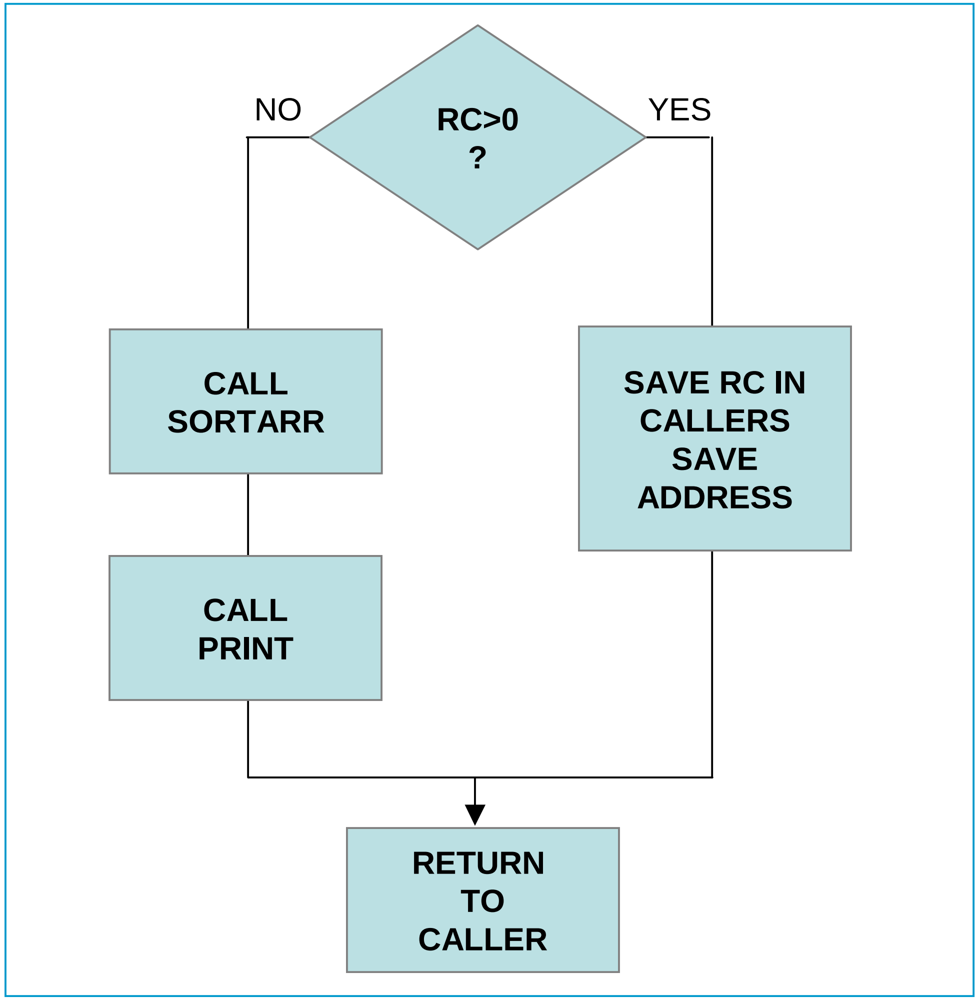
This example shows the remaining half of the coding for main module.

Continued…

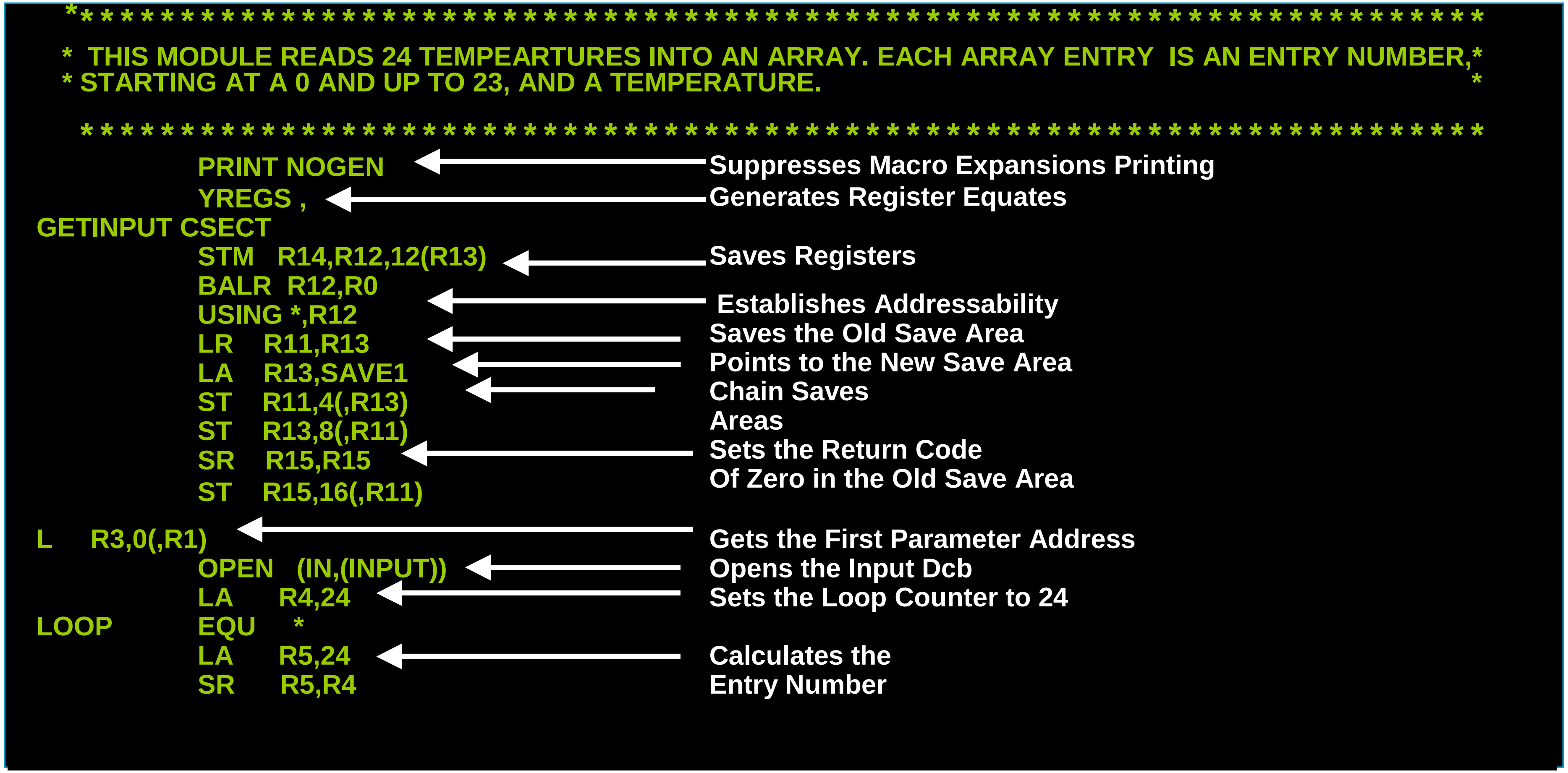
The Main Module (cont'd)

Control is then passed to the GETINPUT module, and on return, the return code is tested. If it is non-zero, it is placed into the caller's save area, and return control to the caller.

Otherwise, control is successively passed on to the SORTARR and PRINT modules, and then return control to the caller.



The GETINPUT Module



```
*******************************************************************************
*  THIS MODULE READS 24 TEMPEARTURES INTO AN ARRAY. EACH ARRAY ENTRY  IS AN ENTRY NUMBER,*
* STARTING AT A 0 AND UP TO 23, AND A TEMPERATURE.                                        *

 ******************************************************************************
              PRINT NOGEN
              YREGS ,
GETINPUT CSECT
              STM   R14,R12,12(R13)
              BALR  R12,R0
              USING *,R12
              LR    R11,R13
              LA    R13,SAVE1
              ST    R11,4(,R13)
              ST    R13,8(,R11)
              SR    R15,R15
              ST    R15,16(,R11)

L     R3,0(,R1)
              OPEN   (IN,(INPUT))
              LA     R4,24
LOOP          EQU    *
              LA     R5,24
              SR     R5,R4
```

Suppresses Macro Expansions Printing

Generates Register Equates

Saves Registers

Establishes Addressability

Saves the Old Save Area

Points to the New Save Area

Chain Saves
Areas

Sets the Return Code
Of Zero in the Old Save Area

Gets the First Parameter Address

Opens the Input Dcb

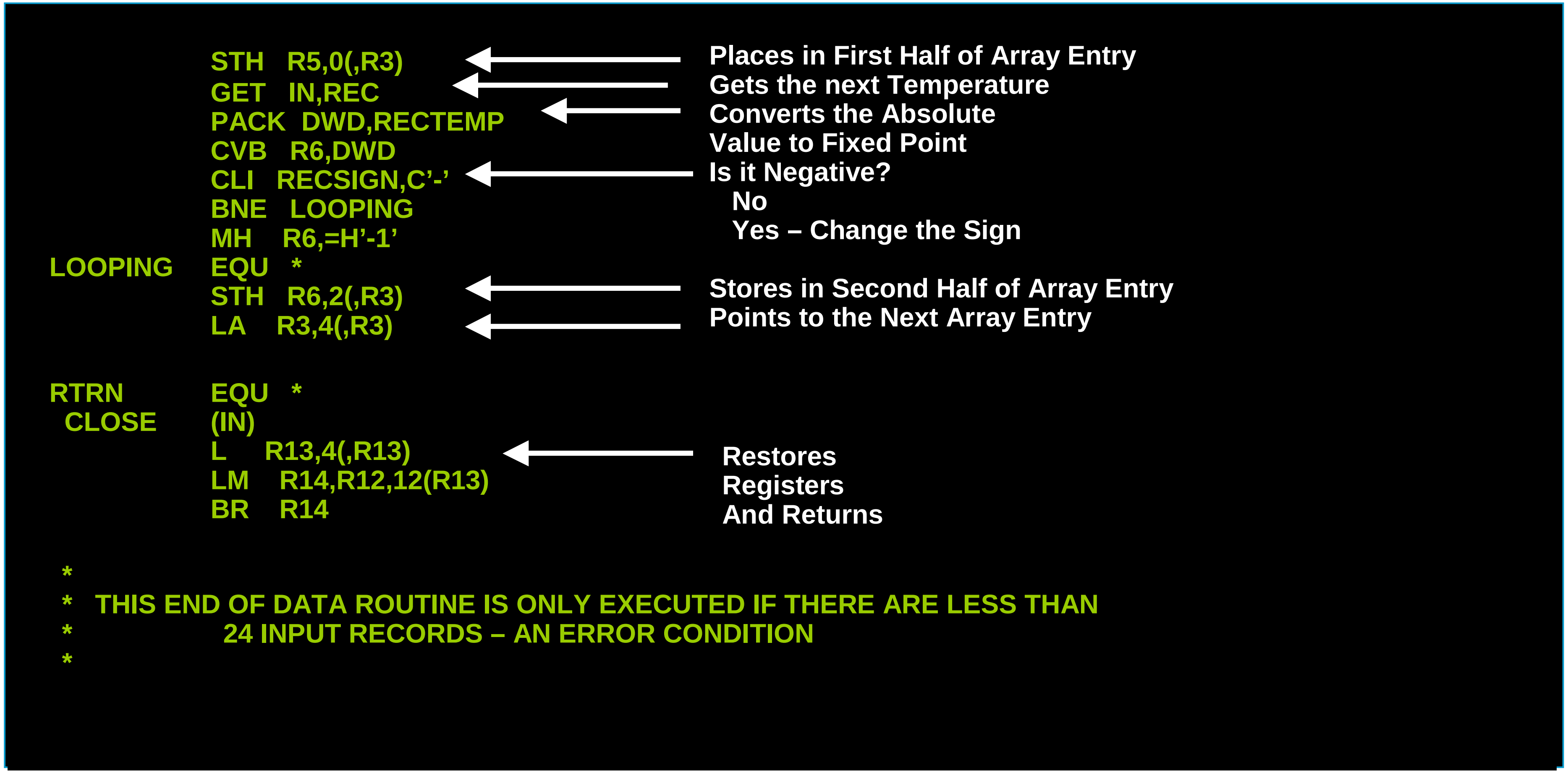
Sets the Loop Counter to 24

Calculates the
Entry Number

A part of the code of GETINPUT module is shown in the example.

Continued…

The GETINPUT Module (cont'd)



```
             STH   R5,0(,R3)
             GET   IN,REC
             PACK  DWD,RECTEMP
             CVB   R6,DWD
             CLI   RECSIGN,C'-'
             BNE   LOOPING
             MH    R6,=H'-1'
LOOPING      EQU   *
             STH   R6,2(,R3)
             LA    R3,4(,R3)

RTRN         EQU   *
 CLOSE       (IN)
             L     R13,4(,R13)
             LM    R14,R12,12(R13)
             BR    R14

 *
 *  THIS END OF DATA ROUTINE IS ONLY EXECUTED IF THERE ARE LESS THAN
 *                24 INPUT RECORDS – AN ERROR CONDITION
 *
```

Places in First Half of Array Entry
Gets the next Temperature
Converts the Absolute
Value to Fixed Point
Is it Negative?
No
Yes – Change the Sign

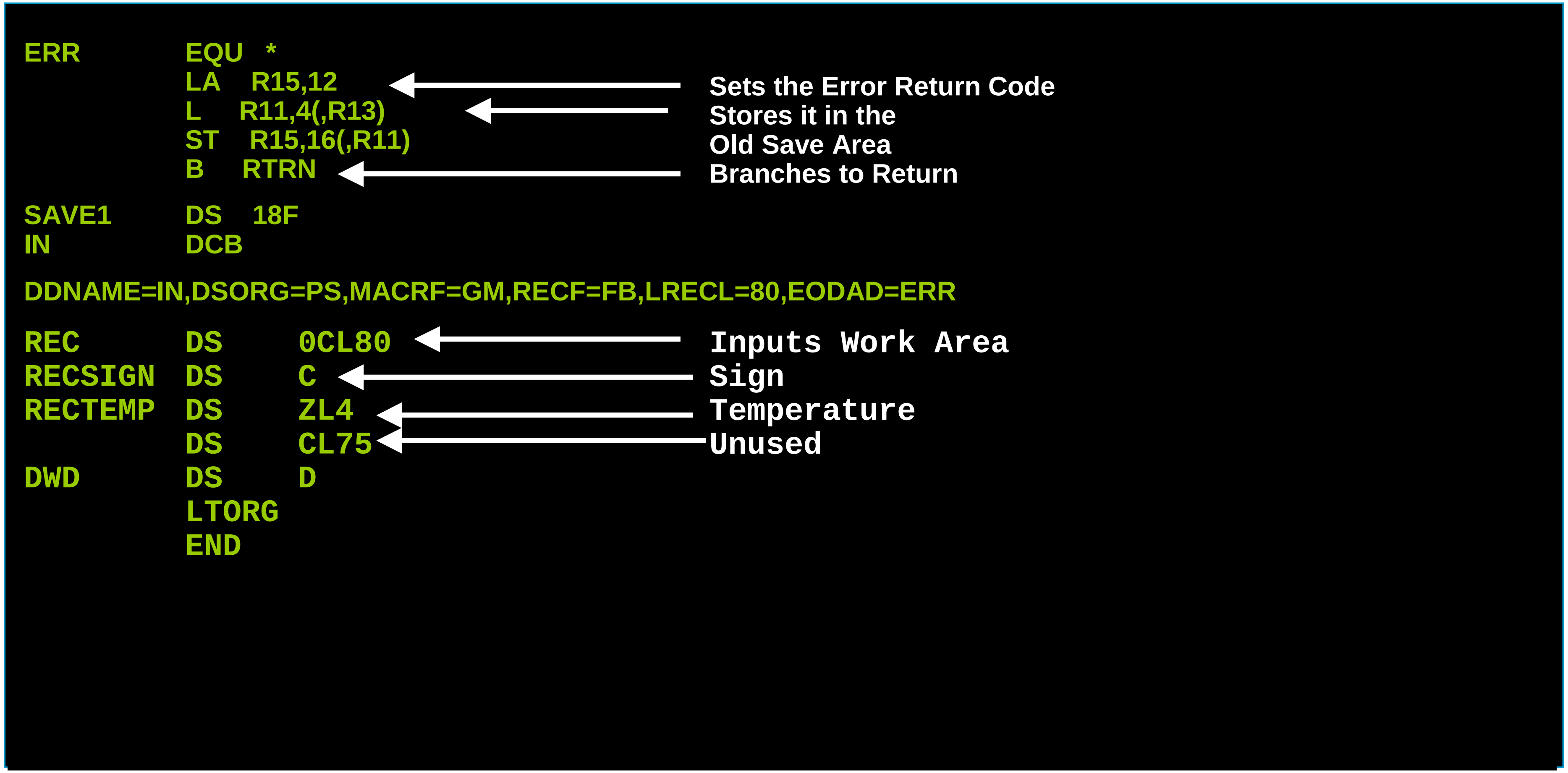
Stores in Second Half of Array Entry
Points to the Next Array Entry

Restores
Registers
And Returns

Another part of the code of GETINPUT module is shown in the example.

Continued…

The GETINPUT Module (cont'd)



```
ERR      EQU  *
         LA   R15,12
         L    R11,4(,R13)
         ST   R15,16(,R11)
         B    RTRN
SAVE1    DS   18F
IN       DCB
DDNAME=IN,DSORG=PS,MACRF=GM,RECF=FB,LRECL=80,EODAD=ERR
REC      DS     0CL80
RECSIGN  DS     C
RECTEMP  DS     ZL4
         DS     CL75
DWD      DS     D
         LTORG
         END
```

Sets the Error Return Code
Stores it in the
Old Save Area
Branches to Return

Inputs Work Area
Sign
Temperature
Unused

The remaining part of the code of GETINPUT module is shown in the example.

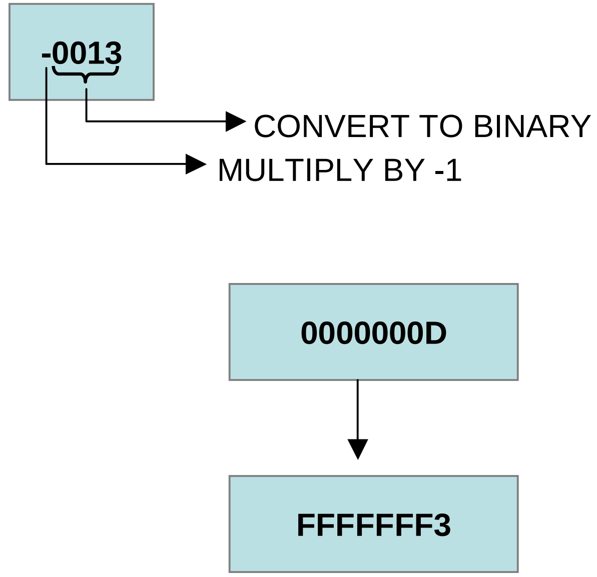
Continued…

The GETINPUT Module (cont'd)

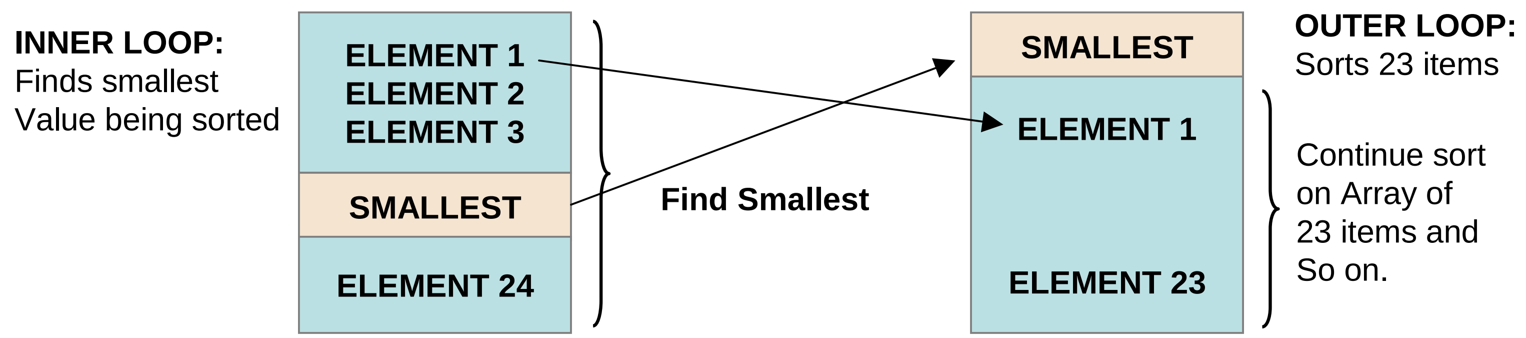
To generate the entry number for each array entry, the value of the loop counter in R4 is subtracted from 24. The value in R4 decreases from 24 to 1, so the entry number will range from 0 to 23.

To handle negative numbers on input, the absolute value of the temperature is converted to a fixed point. If the sign is negative, the fixed-point value is multiplied by -1.

The parameter passed to this routine is the address of the array, and each array is 4 bytes in length. Therefore, the array can be processed by incrementing R3 by 4, each time through the loop.



The SORTARR Module



SORTARR module sorts the array by using a selection sort algorithm. This algorithm looks at all the elements of the array, and selects the smallest. It then swaps the smallest value, with the value in the first position. When the first element of the sorted array is in place, the procedure is repeated for the array, one position smaller, which starts at the next element.

This is the outer loop of the code, controlled by R4. Within the outer loop, it is assumed that the first item is the smallest. Each of the other items are then compared and if one smaller than the current smallest one is found, its value and location is saved. After all items have been compared and the smallest item is found, it should be swapped with the first. This inner loop is controlled by R6.

Continued…

The SORTARR Module (cont'd)

Swapping is done without an intermediate work area, using EXCLUSIVE OR.

R9 is used to hold the value of the currently lowest entry, and R8 holds the address of that entry.

As for the other registers, they perform the following functions:

R3: It holds the address of First Unsorted Array Element

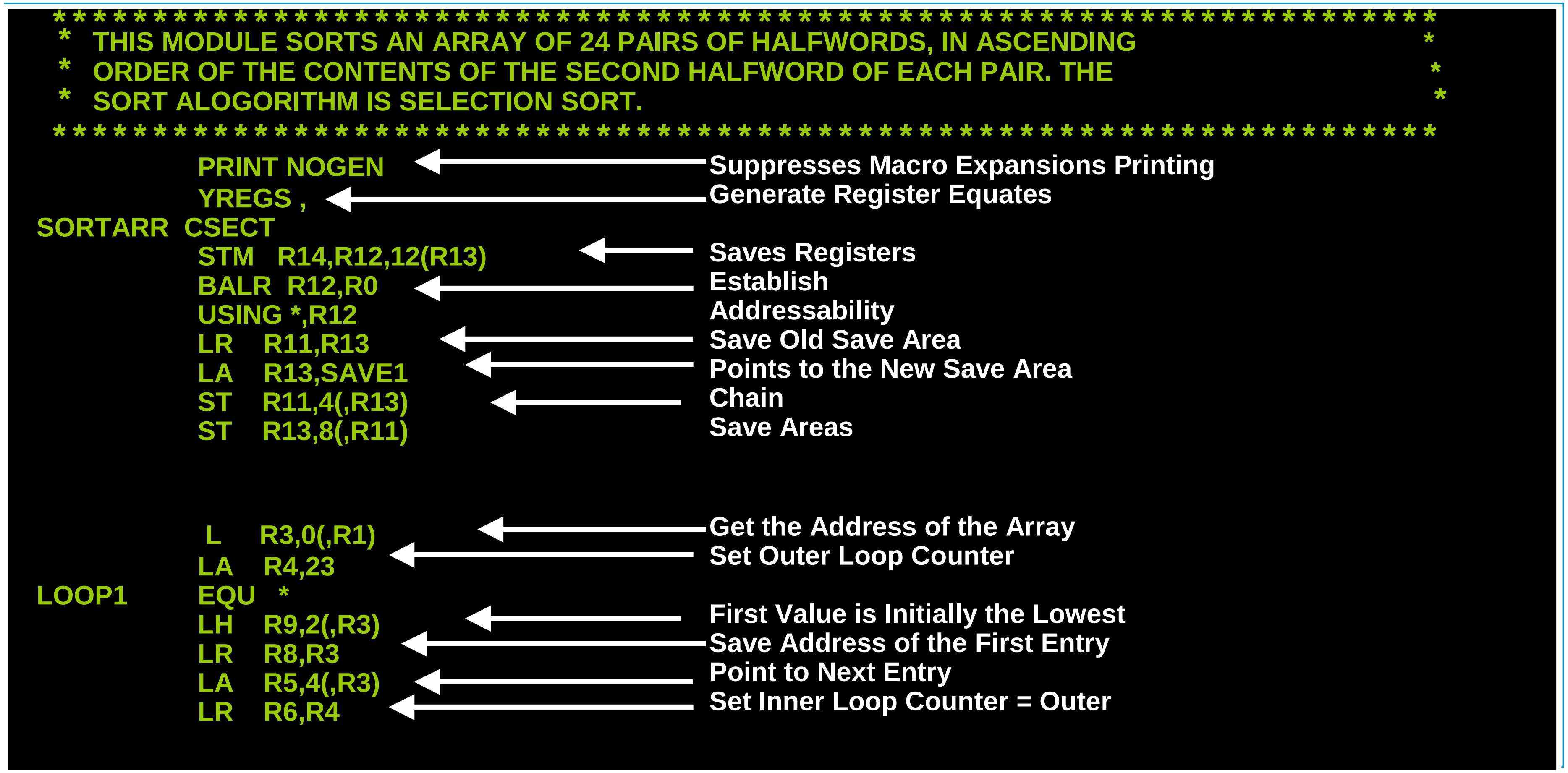
R4: It holds the Counter for Outer Loop (starts at 23)

R5: It holds the address of Current Array Entry being Checked

R6: It holds the Counter for Inner Loop (start = Outer Loop Counter)

Continued…

The SORTARR Module (cont’d)

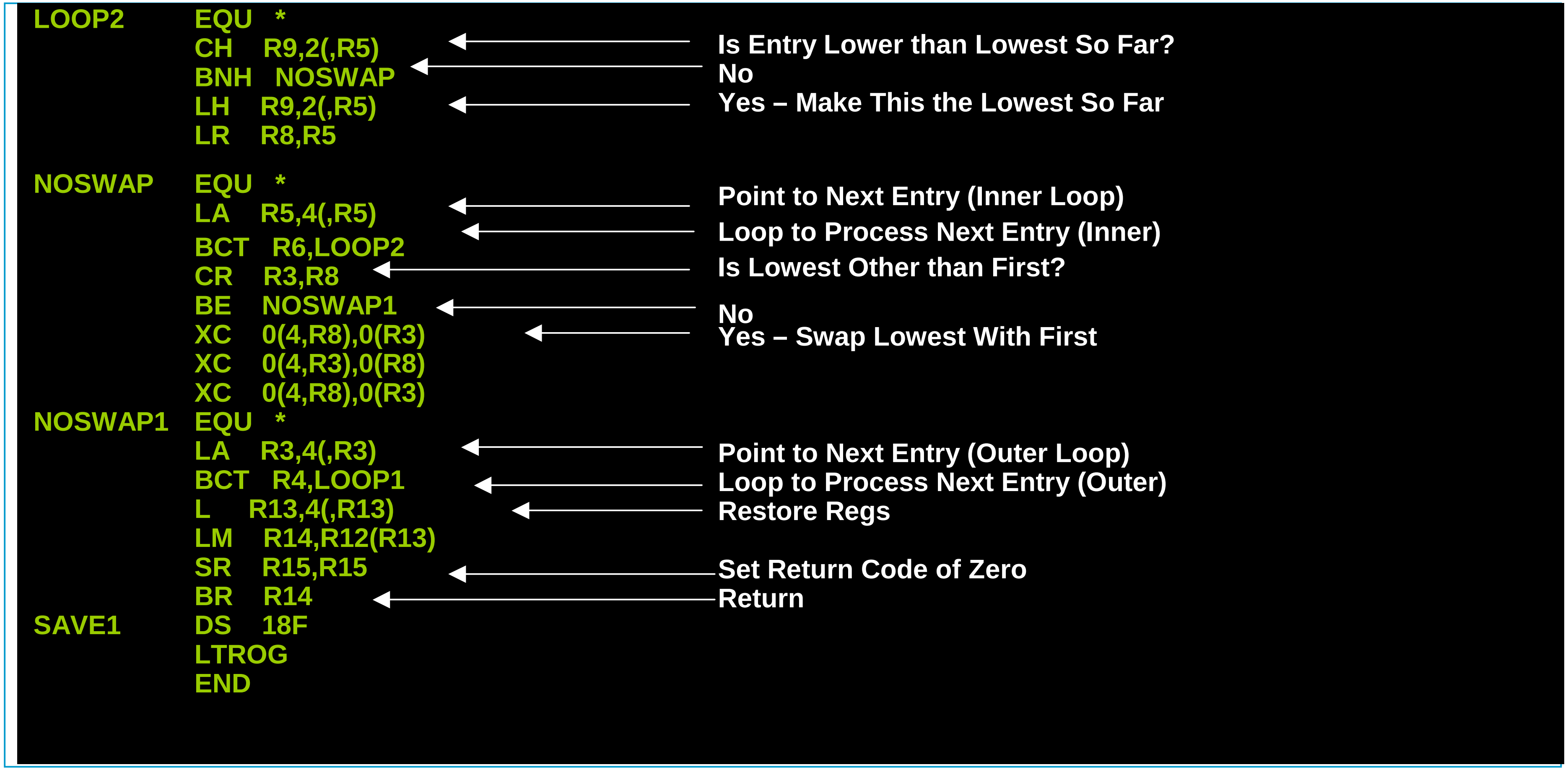


```
*********************************************************************
 *  THIS MODULE SORTS AN ARRAY OF 24 PAIRS OF HALFWORDS, IN ASCENDING   *
 *  ORDER OF THE CONTENTS OF THE SECOND HALFWORD OF EACH PAIR. THE      *
 *  SORT ALOGORITHM IS SELECTION SORT.                                  *
*********************************************************************
         PRINT NOGEN                    Suppresses Macro Expansions Printing
         YREGS ,                        Generate Register Equates
SORTARR  CSECT
         STM   R14,R12,12(R13)          Saves Registers
         BALR  R12,R0                   Establish
         USING *,R12                    Addressability
         LR    R11,R13                  Save Old Save Area
         LA    R13,SAVE1                Points to the New Save Area
         ST    R11,4(,R13)              Chain
         ST    R13,8(,R11)              Save Areas


          L    R3,0(,R1)                Get the Address of the Array
         LA    R4,23                    Set Outer Loop Counter
LOOP1    EQU   *
         LH    R9,2(,R3)                First Value is Initially the Lowest
         LR    R8,R3                    Save Address of the First Entry
         LA    R5,4(,R3)                Point to Next Entry
         LR    R6,R4                    Set Inner Loop Counter = Outer
```

A part of the code of the SORTARR module is shown in the example.

Continued…

The SORTARR Module (cont’d)



```
LOOP2     EQU  *
          CH   R9,2(,R5)          Is Entry Lower than Lowest So Far?
          BNH  NOSWAP             No
          LH   R9,2(,R5)          Yes – Make This the Lowest So Far
          LR   R8,R5

NOSWAP    EQU  *
          LA   R5,4(,R5)          Point to Next Entry (Inner Loop)
          BCT  R6,LOOP2           Loop to Process Next Entry (Inner)
          CR   R3,R8              Is Lowest Other than First?
          BE   NOSWAP1            No
          XC   0(4,R8),0(R3)      Yes – Swap Lowest With First
          XC   0(4,R3),0(R8)
          XC   0(4,R8),0(R3)
NOSWAP1   EQU  *
          LA   R3,4(,R3)          Point to Next Entry (Outer Loop)
          BCT  R4,LOOP1           Loop to Process Next Entry (Outer)
          L    R13,4(,R13)        Restore Regs
          LM   R14,R12(R13)
          SR   R15,R15            Set Return Code of Zero
          BR   R14                Return
SAVE1     DS   18F
          LTROG
          END
```

The remaining part of the code of the SORTARR module is shown in the example.

The PRINT Module

The print module is the longest in the program. It determines the following four things:

- Lowest temperature
- Highest temperature
- Average temperature
- Median temperature

It first gets the lowest temperature. The array has been sorted into ascending order by temperature, so the fist entry is the lowest. The temperature is edited into the output line, and then the corresponding time is added.

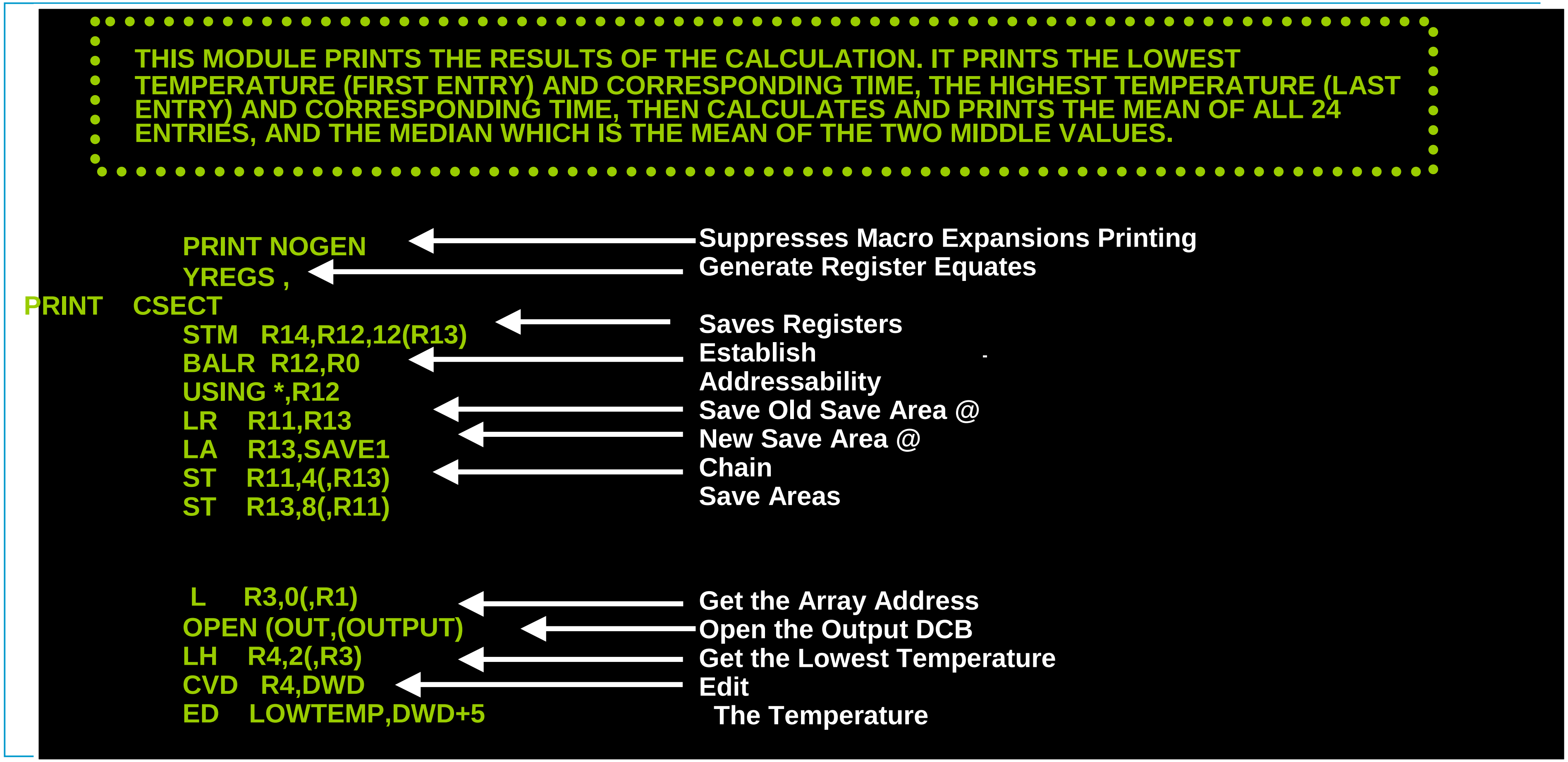
The highest temperature is also processed in a similar manner.

The average temperature is determined by adding all 24 temperatures, and dividing by 24, and then rounding.

The median temperature is the average of the two middle entries in the array.

Continued…

The PRINT Module (cont'd)



THIS MODULE PRINTS THE RESULTS OF THE CALCULATION. IT PRINTS THE LOWEST TEMPERATURE (FIRST ENTRY) AND CORRESPONDING TIME, THE HIGHEST TEMPERATURE (LAST ENTRY) AND CORRESPONDING TIME, THEN CALCULATES AND PRINTS THE MEAN OF ALL 24 ENTRIES, AND THE MEDIAN WHICH IS THE MEAN OF THE TWO MIDDLE VALUES.

```
         PRINT NOGEN
         YREGS ,
PRINT    CSECT
         STM   R14,R12,12(R13)
         BALR  R12,R0
         USING *,R12
         LR    R11,R13
         LA    R13,SAVE1
         ST    R11,4(,R13)
         ST    R13,8(,R11)


          L    R3,0(,R1)
         OPEN (OUT,(OUTPUT)
         LH    R4,2(,R3)
         CVD   R4,DWD
         ED    LOWTEMP,DWD+5
```

Suppresses Macro Expansions Printing

Generate Register Equates

Saves Registers

Establish
Addressability

Save Old Save Area @

New Save Area @

Chain
Save Areas

Get the Array Address

Open the Output DCB

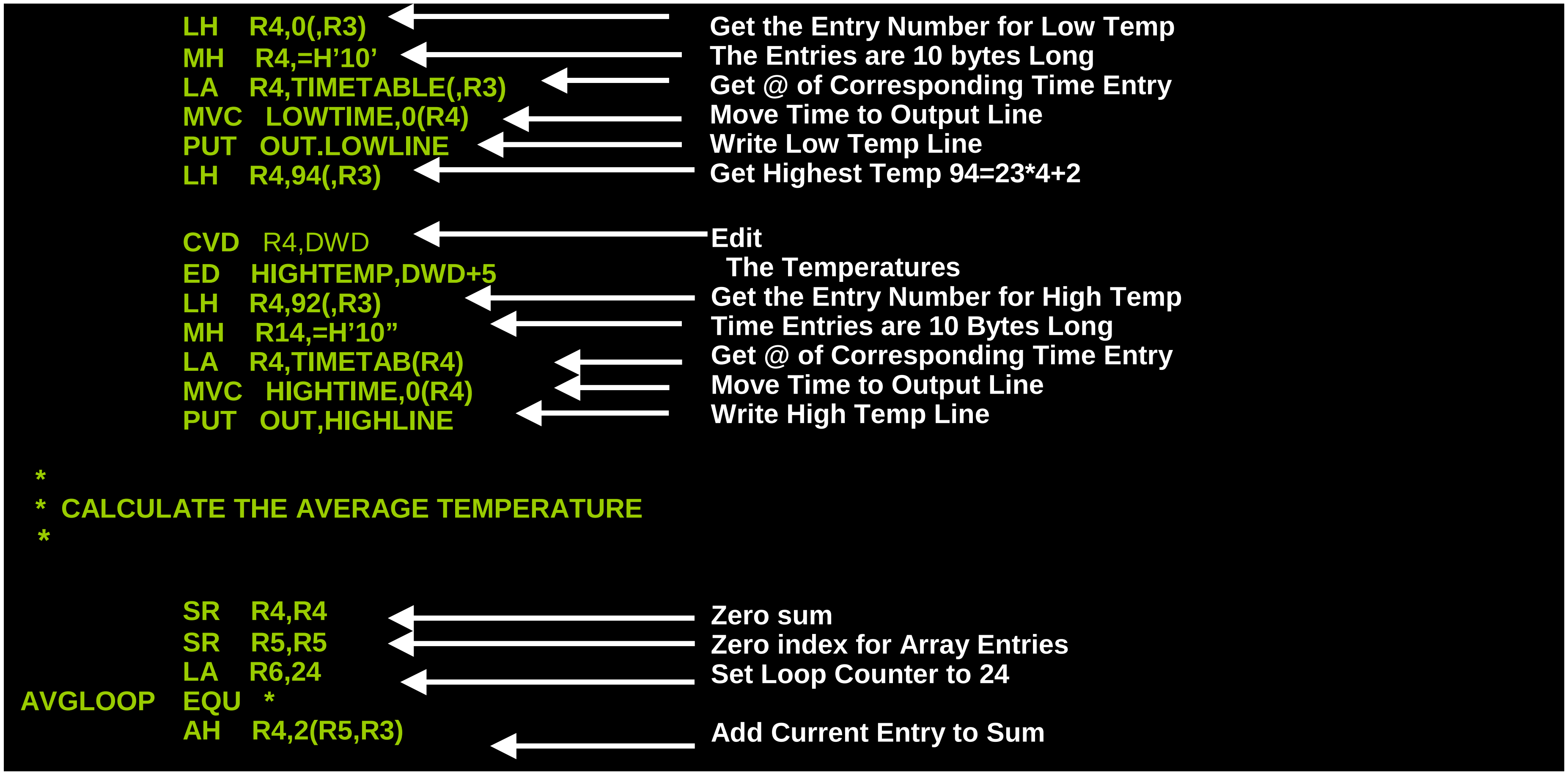
Get the Lowest Temperature

Edit
The Temperature

Like other modules, PRINT module too starts with a block of comments explaining its purpose. The code of PRINT module is shown in the example.

Continued…

The PRINT Module (cont’d)

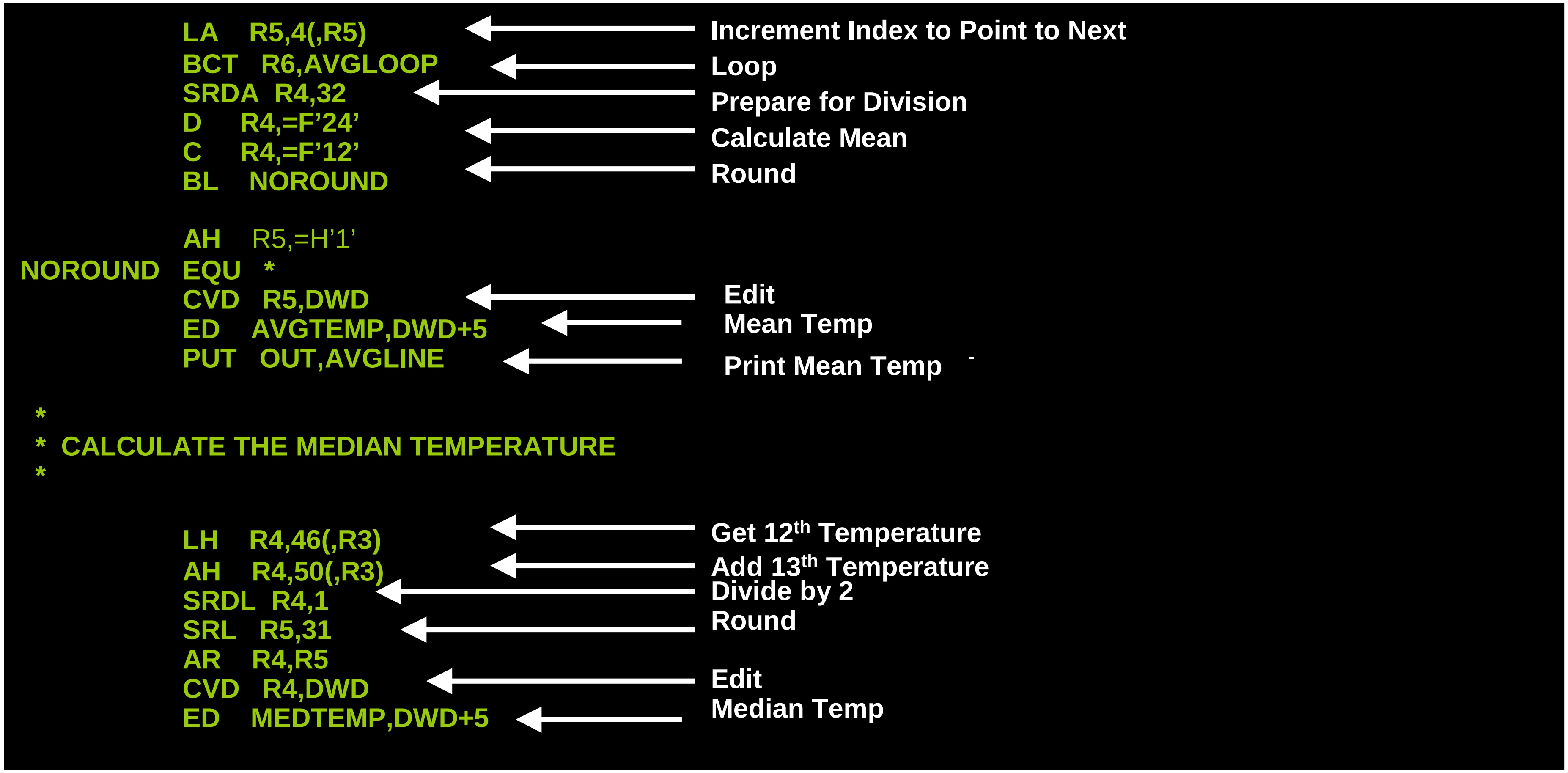


```
         LH    R4,0(,R3)             Get the Entry Number for Low Temp
         MH    R4,=H’10’             The Entries are 10 bytes Long
         LA    R4,TIMETABLE(,R3)     Get @ of Corresponding Time Entry
         MVC   LOWTIME,0(R4)         Move Time to Output Line
         PUT   OUT.LOWLINE           Write Low Temp Line
         LH    R4,94(,R3)            Get Highest Temp 94=23*4+2

         CVD   R4,DWD                Edit
         ED    HIGHTEMP,DWD+5          The Temperatures
         LH    R4,92(,R3)            Get the Entry Number for High Temp
         MH    R14,=H’10”            Time Entries are 10 Bytes Long
         LA    R4,TIMETAB(R4)        Get @ of Corresponding Time Entry
         MVC   HIGHTIME,0(R4)        Move Time to Output Line
         PUT   OUT,HIGHLINE          Write High Temp Line

*
*  CALCULATE THE AVERAGE TEMPERATURE
*

         SR    R4,R4                 Zero sum
         SR    R5,R5                 Zero index for Array Entries
         LA    R6,24                 Set Loop Counter to 24
AVGLOOP  EQU   *
         AH    R4,2(R5,R3)           Add Current Entry to Sum
```

The continued code of PRINT module is shown in the example.

Continued…

The PRINT Module (cont’d)

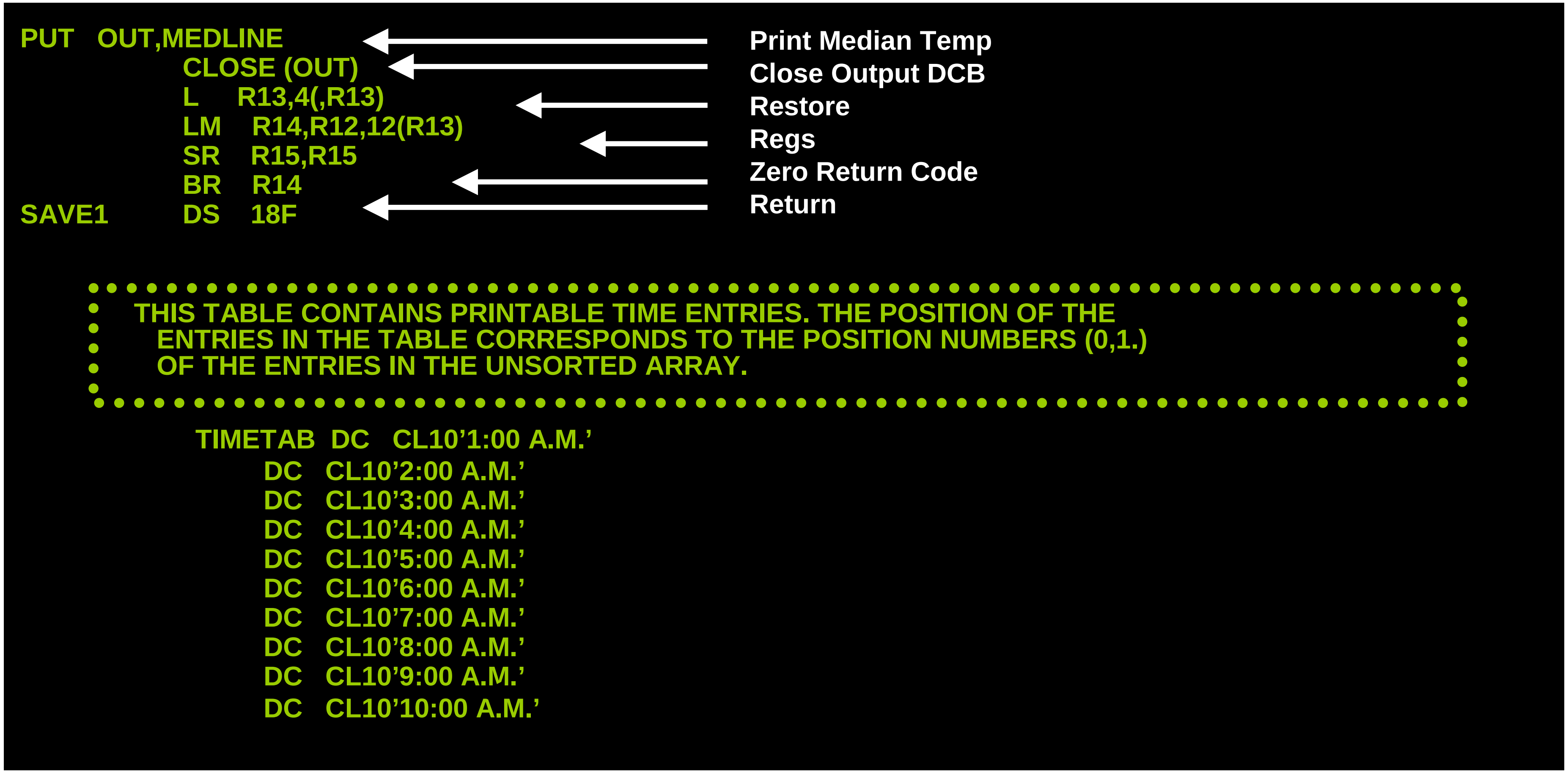


```
         LA    R5,4(,R5)
         BCT   R6,AVGLOOP
         SRDA  R4,32
         D     R4,=F’24’
         C     R4,=F’12’
         BL    NOROUND

         AH    R5,=H’1’
NOROUND  EQU   *
         CVD   R5,DWD
         ED    AVGTEMP,DWD+5
         PUT   OUT,AVGLINE

*
*  CALCULATE THE MEDIAN TEMPERATURE
*

         LH    R4,46(,R3)
         AH    R4,50(,R3)
         SRDL  R4,1
         SRL   R5,31
         AR    R4,R5
         CVD   R4,DWD
         ED    MEDTEMP,DWD+5
```

The continued code of PRINT module is shown in the example.

Continued…

The PRINT Module (cont’d)



```
PUT   OUT,MEDLINE
           CLOSE (OUT)
           L    R13,4(,R13)
           LM   R14,R12,12(R13)
           SR   R15,R15
           BR   R14
SAVE1      DS   18F
```

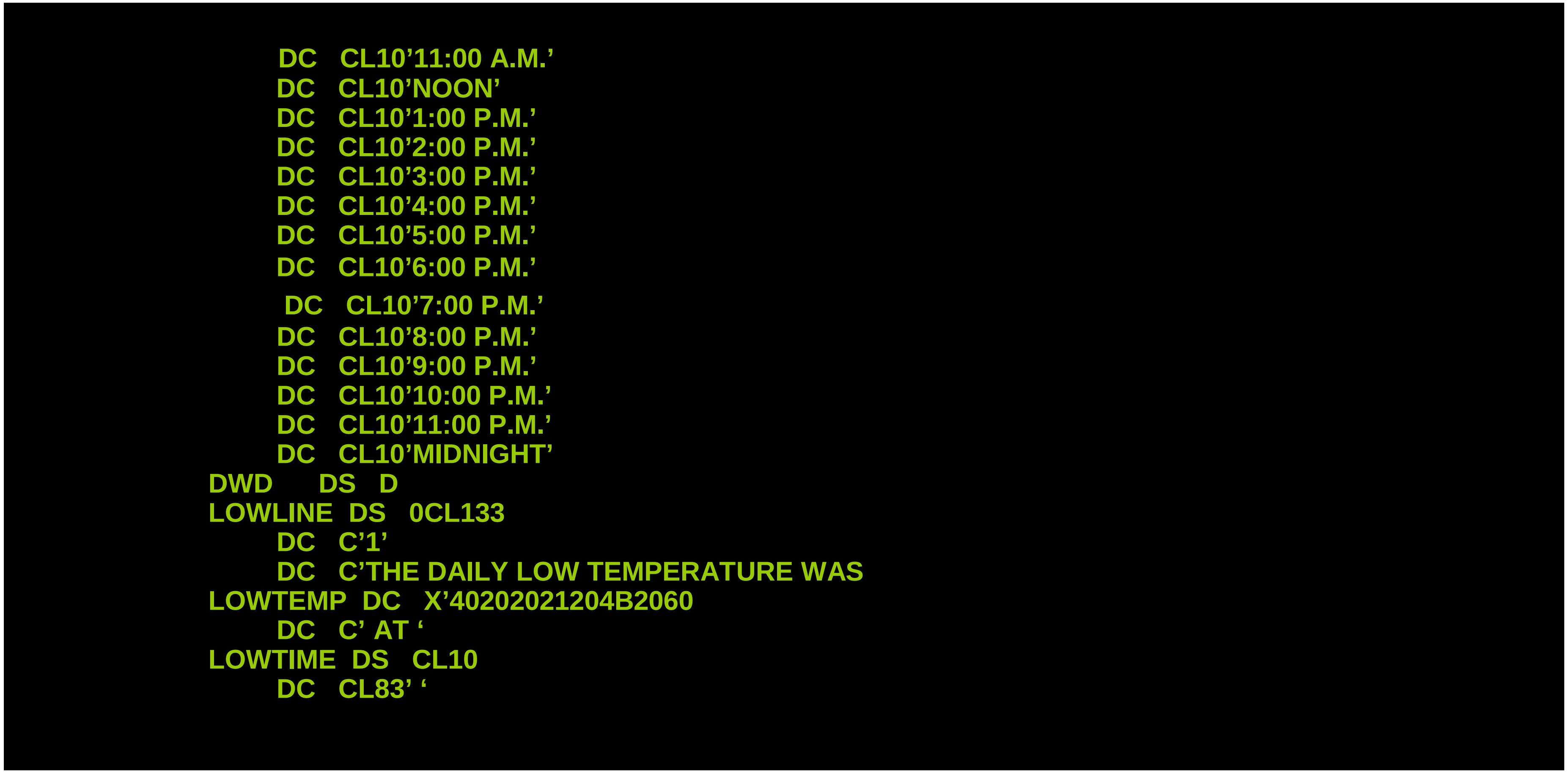
Print Median Temp
Close Output DCB
Restore
Regs
Zero Return Code
Return

```
   THIS TABLE CONTAINS PRINTABLE TIME ENTRIES. THE POSITION OF THE
     ENTRIES IN THE TABLE CORRESPONDS TO THE POSITION NUMBERS (0,1.)
     OF THE ENTRIES IN THE UNSORTED ARRAY.

            TIMETAB  DC   CL10’1:00 A.M.’
                     DC   CL10’2:00 A.M.’
                     DC   CL10’3:00 A.M.’
                     DC   CL10’4:00 A.M.’
                     DC   CL10’5:00 A.M.’
                     DC   CL10’6:00 A.M.’
                     DC   CL10’7:00 A.M.’
                     DC   CL10’8:00 A.M.’
                     DC   CL10’9:00 A.M.’
                     DC   CL10’10:00 A.M.’
```

The continued code of the PRINT module is shown in the example.

Continued…

The PRINT Module (cont’d)

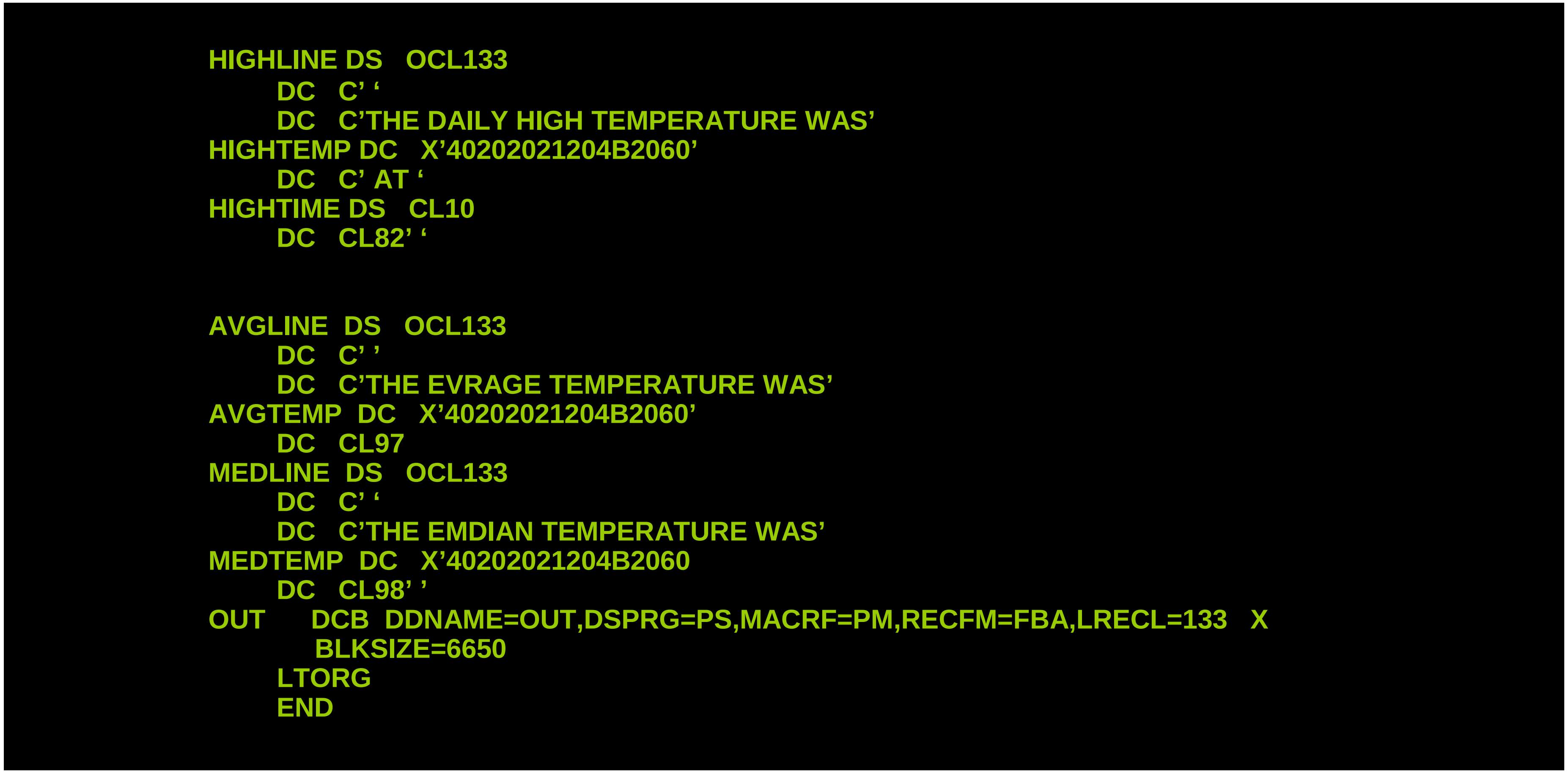


```
          DC   CL10’11:00 A.M.’
          DC   CL10’NOON’
          DC   CL10’1:00 P.M.’
          DC   CL10’2:00 P.M.’
          DC   CL10’3:00 P.M.’
          DC   CL10’4:00 P.M.’
          DC   CL10’5:00 P.M.’
          DC   CL10’6:00 P.M.’
          DC   CL10’7:00 P.M.’
          DC   CL10’8:00 P.M.’
          DC   CL10’9:00 P.M.’
          DC   CL10’10:00 P.M.’
          DC   CL10’11:00 P.M.’
          DC   CL10’MIDNIGHT’
DWD       DS   D
LOWLINE   DS   0CL133
          DC   C’1’
          DC   C’THE DAILY LOW TEMPERATURE WAS
LOWTEMP   DC   X’40202021204B2060
          DC   C’ AT ‘
LOWTIME   DS   CL10
          DC   CL83’ ‘
```

The continued code of the PRINT module is shown in the example.

Continued…

The PRINT Module (cont’d)



```
HIGHLINE DS   OCL133
         DC   C’ ‘
         DC   C’THE DAILY HIGH TEMPERATURE WAS’
HIGHTEMP DC   X’40202021204B2060’
         DC   C’ AT ‘
HIGHTIME DS   CL10
         DC   CL82’ ‘


AVGLINE  DS   OCL133
         DC   C’ ’
         DC   C’THE EVRAGE TEMPERATURE WAS’
AVGTEMP  DC   X’40202021204B2060’
         DC   CL97
MEDLINE  DS   OCL133
         DC   C’ ‘
         DC   C’THE EMDIAN TEMPERATURE WAS’
MEDTEMP  DC   X’40202021204B2060
         DC   CL98’ ’
OUT       DCB  DDNAME=OUT,DSPRG=PS,MACRF=PM,RECFM=FBA,LRECL=133   X
               BLKSIZE=6650
         LTORG
         END
```

The continued code of the PRINT module is shown in the example.

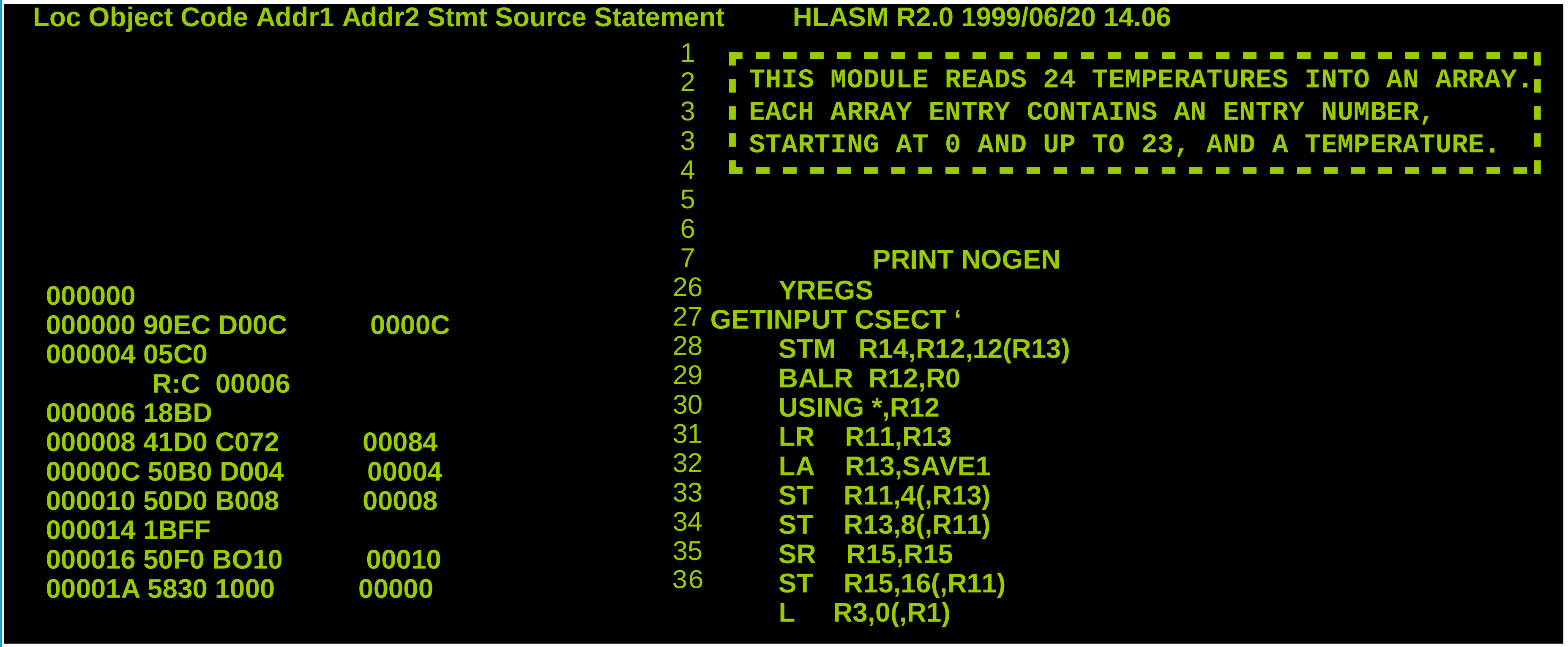
Methods

Under Time Sharing Option/ Extended (TSO/E), the following two processing techniques can be used to assemble and link programs:

- Background Processing: Here the necessary Job Control Language (JCL) is used to perform the assemble, and link it to the source files with the Assembler code. It is then submitted for batch processing. Alternatively, the programmer can put JCL in a separate file, and point to the source code in the DD statements, specifying the assembler input.

- Foreground Processing: Here the programmer can use either TSO/E assemble and link commands directly, or the foreground processing dialogs provided in Interactive System Productivity Facility (ISPF).

Listings



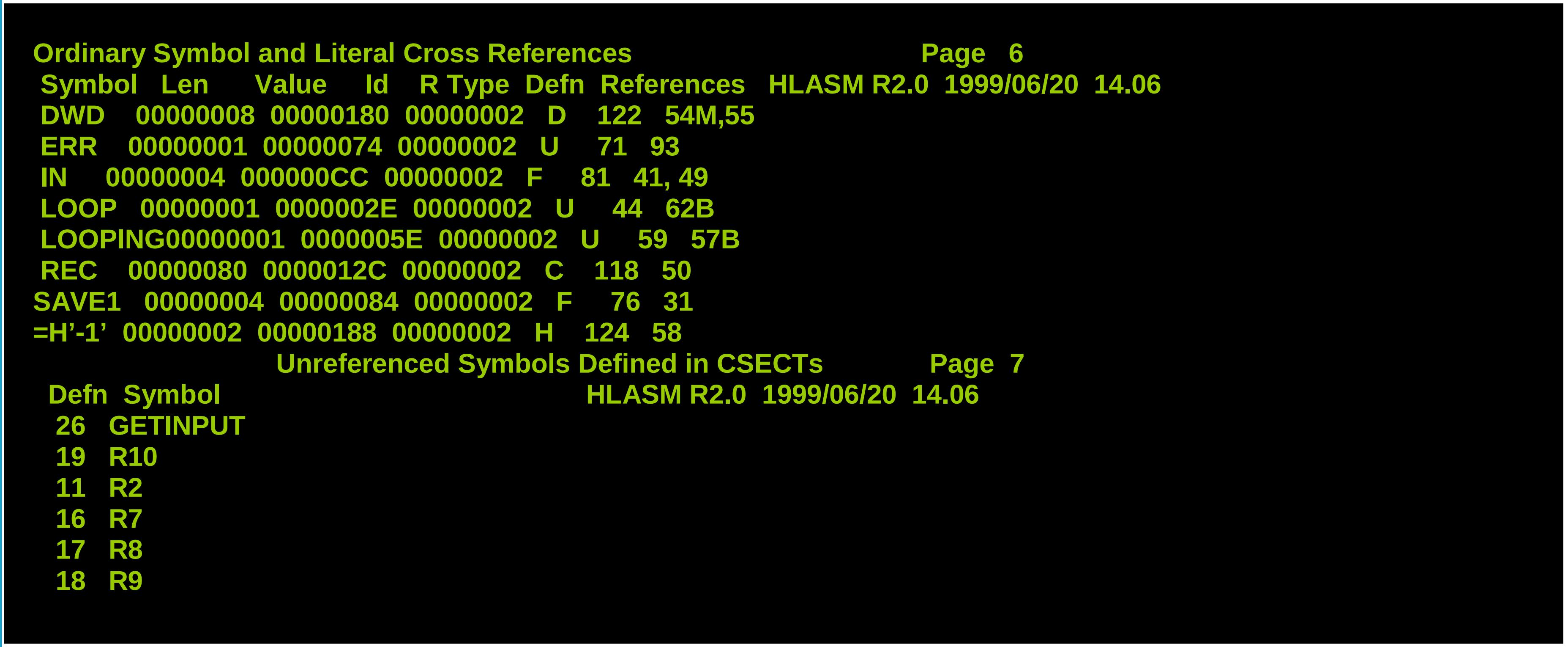
```
  Loc Object Code Addr1 Addr2 Stmt Source Statement          HLASM R2.0 1999/06/20 14.06
                                     1  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                     2  | THIS MODULE READS 24 TEMPERATURES INTO AN ARRAY.|
                                     3  | EACH ARRAY ENTRY CONTAINS AN ENTRY NUMBER,      |
                                     3  | STARTING AT 0 AND UP TO 23, AND A TEMPERATURE.  |
                                     4  - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                     5
                                     6
                                     7               PRINT NOGEN
   000000                           26         YREGS
   000000 90EC D00C       0000C     27 GETINPUT CSECT ‘
   000004 05C0                      28         STM    R14,R12,12(R13)
              R:C  00006            29         BALR  R12,R0
   000006 18BD                      30         USING *,R12
   000008 41D0 C072       00084     31         LR     R11,R13
   00000C 50B0 D004       00004     32         LA     R13,SAVE1
   000010 50D0 B008       00008     33         ST     R11,4(,R13)
   000014 1BFF                      34         ST     R13,8(,R11)
   000016 50F0 BO10       00010     35         SR     R15,R15
   00001A 5830 1000       00000     36         ST     R15,16(,R11)
                                               L      R3,0(,R1)
```

Regardless of the processing techniques, Assembler and linker listings need to be produced. These listings provide useful data to help you when you test and debug your program.

A small section of the listing for GETINPUT module is shown in the example.

Continued…

Listings (cont'd)



```
Ordinary Symbol and Literal Cross References                    Page   6
 Symbol   Len      Value     Id    R Type  Defn  References   HLASM R2.0  1999/06/20  14.06
 DWD    00000008  00000180  00000002   D    122   54M,55
 ERR    00000001  00000074  00000002   U     71   93
 IN     00000004  000000CC  00000002   F     81   41, 49
 LOOP   00000001  0000002E  00000002   U     44   62B
 LOOPING00000001  0000005E  00000002   U     59   57B
 REC    00000080  0000012C  00000002   C    118   50
SAVE1   00000004  00000084  00000002   F     76   31
=H'-1'  00000002  00000188  00000002   H    124   58
                 Unreferenced Symbols Defined in CSECTs          Page  7
  Defn  Symbol                                   HLASM R2.0  1999/06/20  14.06
   26   GETINPUT
   19   R10
   11   R2
   16   R7
   17   R8
   18   R9
```

There are other parts of an Assembler listing, such as the cross-reference listing. The cross-reference listing for GETINPUT is shown in the example.

Continued…

Listings (cont'd)

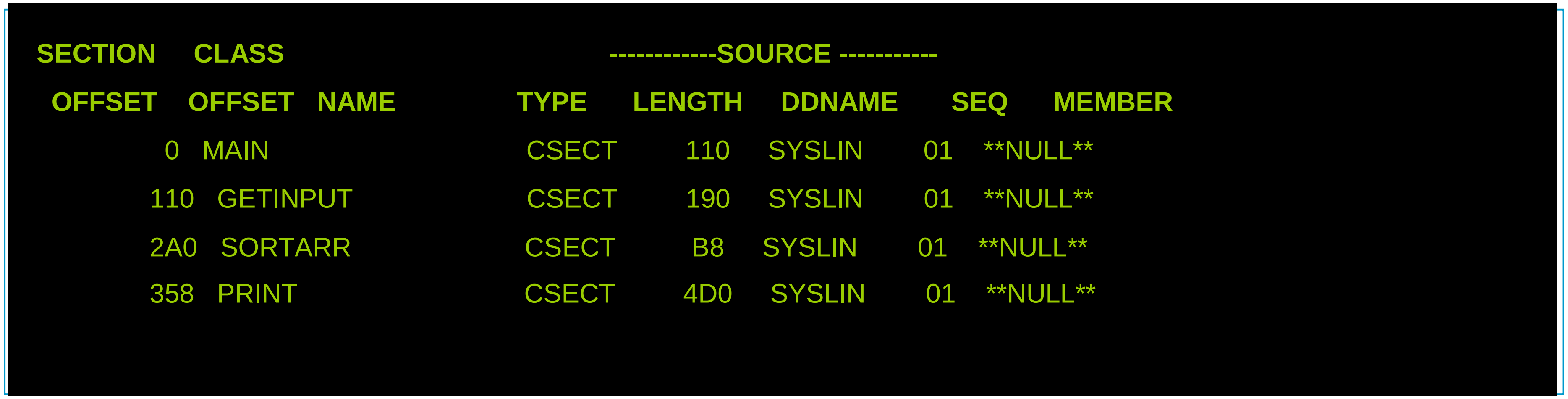
The cross-reference listing shows every symbol defined in your program, as well as where it is defined, and where it is referenced.

The reference entries indicate whether the symbol is modified or not. This information can be very useful in debugging. If a particular variable has the wrong value, you can easily determine the instructions that modify that variable.

To modify program, and to determine a free register to use for calculation, Unreferenced Symbols listing can be used.

Continued…

Listings (cont’d)

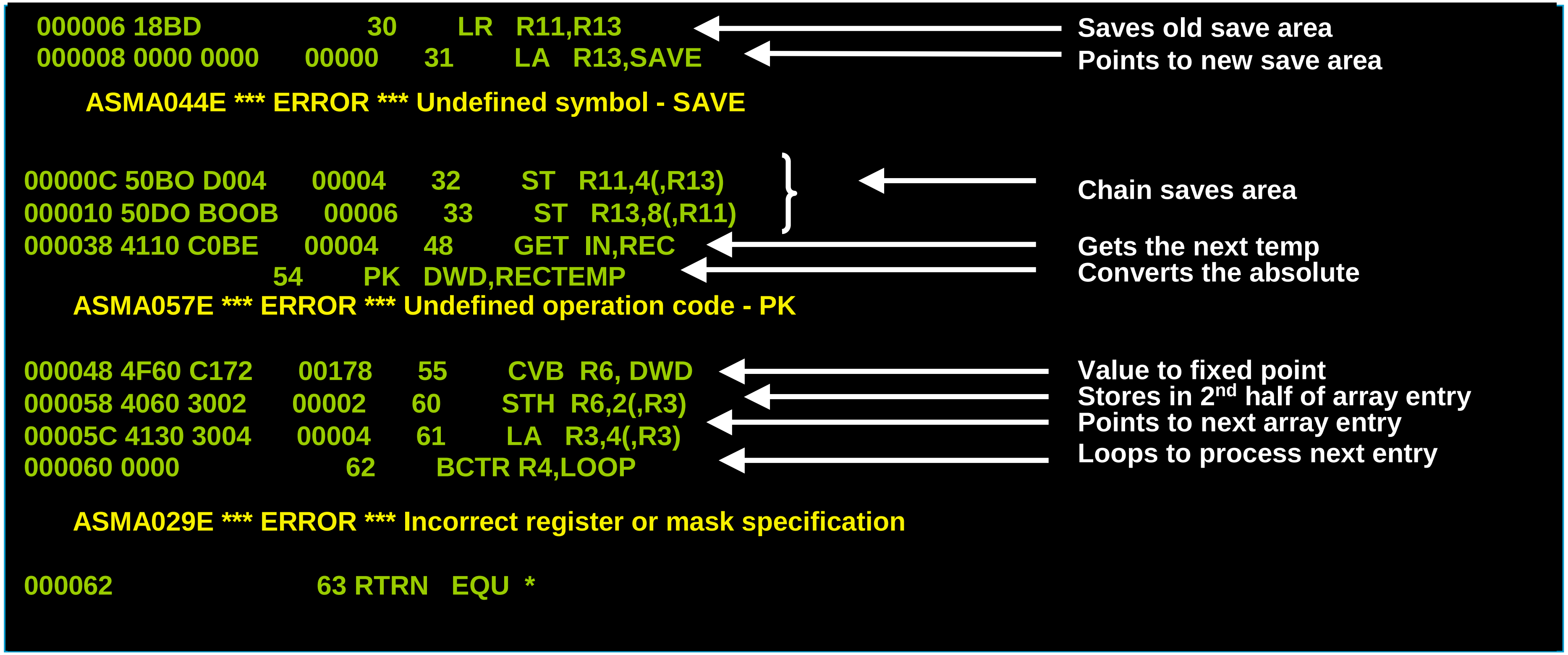


```
SECTION    CLASS                           ------------SOURCE -----------
  OFFSET    OFFSET   NAME            TYPE     LENGTH    DDNAME      SEQ     MEMBER
                 0   MAIN             CSECT        110    SYSLIN       01    **NULL**
               110   GETINPUT         CSECT        190    SYSLIN       01    **NULL**
               2A0   SORTARR          CSECT         B8    SYSLIN       01    **NULL**
               358   PRINT            CSECT        4D0    SYSLIN       01    **NULL**
```

The linkage editor also produces listings, which can be valuable. In the case of the example of temperature, four separately assembled modules were written and then combined with the linkage editor. To determine where the four modules are located, in relation to the beginning of the whole machine language program, the module map will help.

The example here shows a section of the module.

Program Listings



```
 000006 18BD                30      LR   R11,R13
 000008 0000 0000     00000      31       LA   R13,SAVE
        ASMA044E *** ERROR *** Undefined symbol - SAVE

000000C 50BO D004     00004      32       ST   R11,4(,R13)
000010 50DO BOOB      00006      33       ST   R13,8(,R11)
000038 4110 C0BE      00004      48       GET  IN,REC
                               54       PK   DWD,RECTEMP
      ASMA057E *** ERROR *** Undefined operation code - PK

000048 4F60 C172      00178      55       CVB  R6, DWD
000058 4060 3002      00002      60       STH  R6,2(,R3)
00005C 4130 3004      00004      61       LA   R3,4(,R3)
000060 0000                    62       BCTR R4,LOOP

      ASMA029E *** ERROR *** Incorrect register or mask specification

000062                      63 RTRN   EQU  *
```

Saves old save area

Points to new save area

Chain saves area

Gets the next temp

Converts the absolute

Value to fixed point

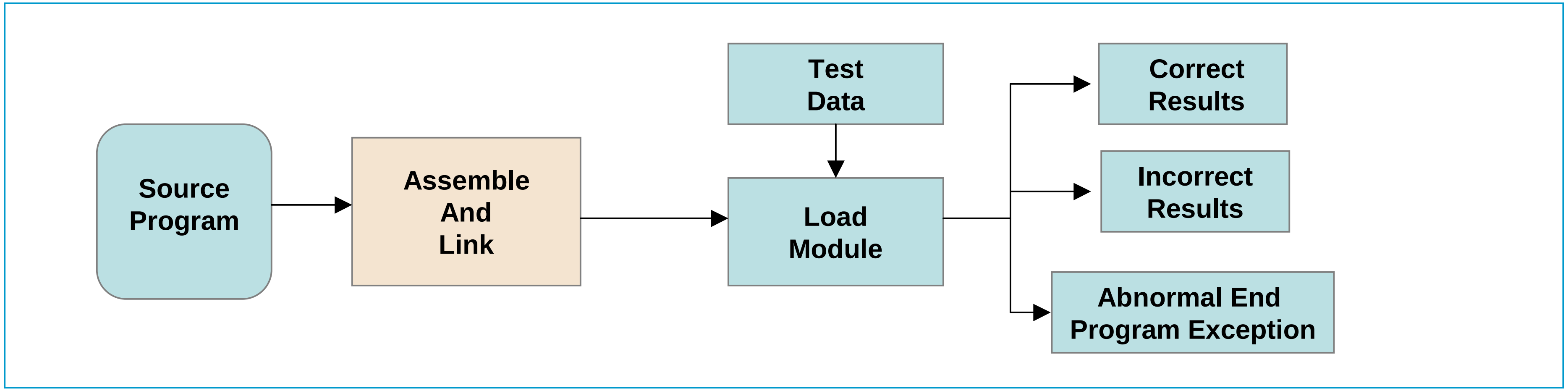
Stores in 2nd half of array entry

Points to next array entry

Loops to process next entry

Messages describing syntax errors appear in the program listing, along with the statement error. In each case, the error message follows the statement in error, and consists of an error number and an error description. The listing for a version of GETINPUT, with some errors, is shown in the example.

Symptom Dump



Once the syntax errors have been removed from a program, you can start the process of testing and debugging. In testing, you run the program using test data, which produces predictable output. When the actual output produced by your program, differs from the expected values, you know there is a problem, and must begin to debug the program.

In some cases, while testing the program, it does not run to completion and terminates with a program exception. In such cases what needs to be determined is at which point in the program the error has occurred, so that the problem can be corrected.

Often the information necessary to diagnose and correct the problem is present in the symptom dump, which is part of the system log for the job.

Continued…

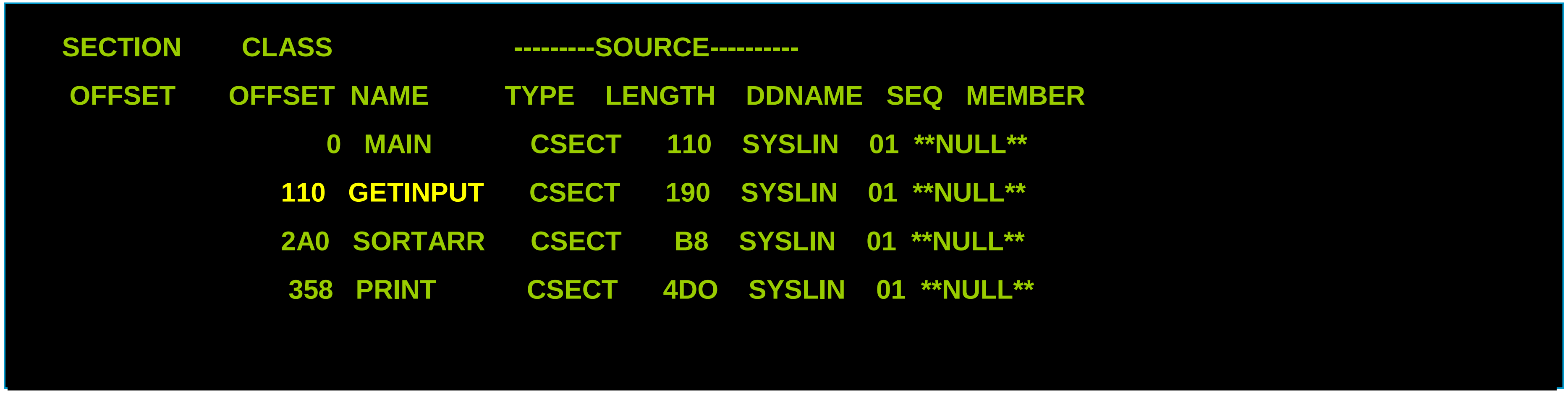
Symptom Dump (cont'd)

Symptom Dump Output

```
 SYSTEM COMPLETION CODE=0c7   REASON CODE=00000007
  TIME=14.18.14   SEQ=00138   CPU=0000   ASID=0015
  PSW AT TIME OF ERROR 078D2000    0000693A   ILC  4     INTC        07
     ACTIVE LOAD MODULE               ADDRESS=000067D8  OFFSET=00000162
     NAME=GO
     DATA AT PSW   00006934  –  C1274F60   C17A9560   C1264770
     GPR  0-3    00000001    00006A14    00000040    00006874
     GPR  4-7    00000018    00000000    009BBFF8    FD000000
     GPR  8-11   009FD080   809DE628   00000000    0000682C
```

The example shows an excerpt from a symptom dump.

Module Map



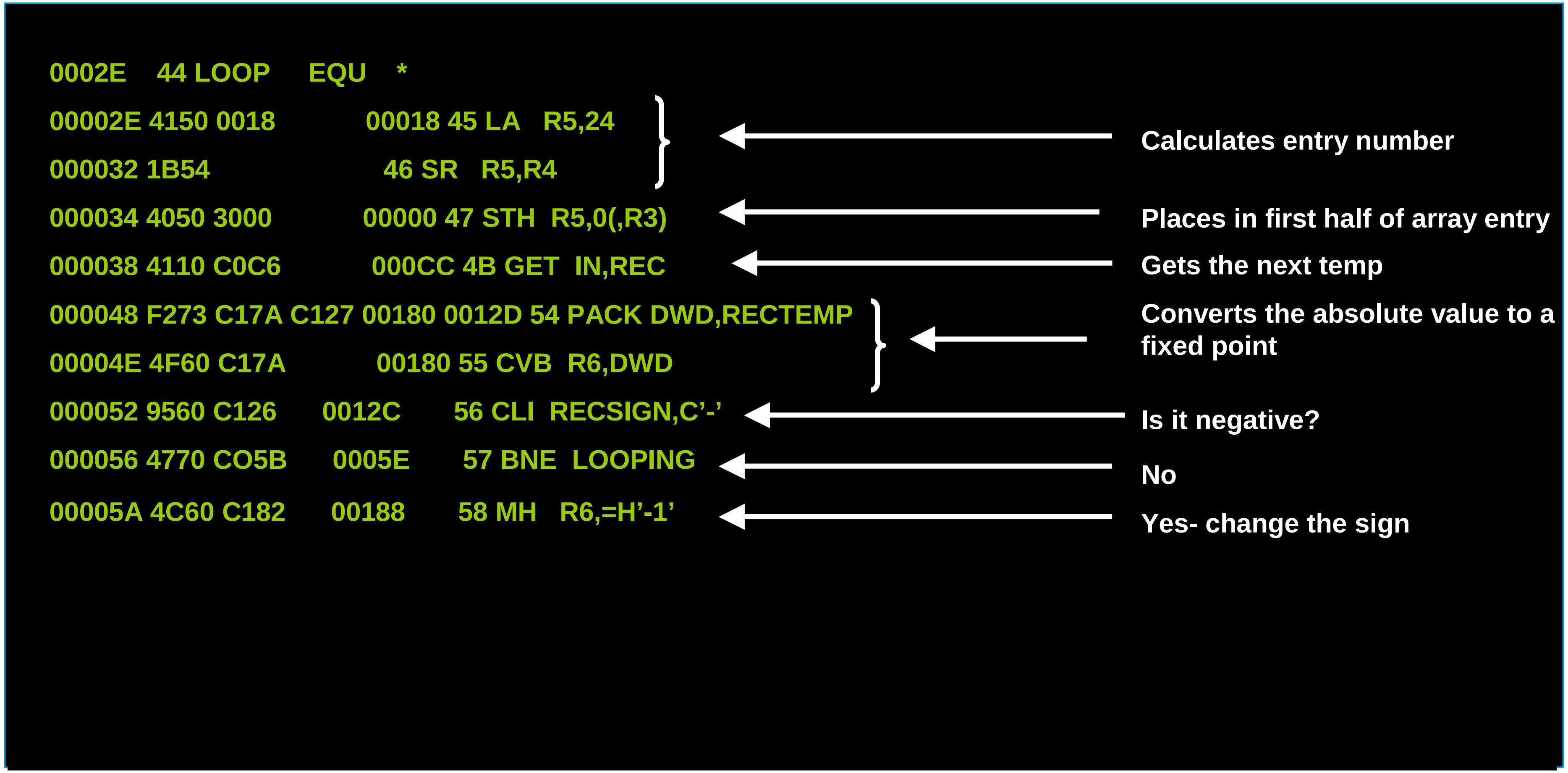
```
SECTION     CLASS              ---------SOURCE----------
 OFFSET     OFFSET  NAME        TYPE   LENGTH   DDNAME   SEQ   MEMBER
                0   MAIN         CSECT    110    SYSLIN   01  **NULL**
              110   GETINPUT     CSECT    190    SYSLIN   01  **NULL**
              2A0   SORTARR      CSECT     B8    SYSLIN   01  **NULL**
              358   PRINT        CSECT    4DO    SYSLIN   01  **NULL**
```

After you view the symptom dump, it is important to take a look at the module map to determine which module the error occurred in, and at what displacement.

The example of a module map, here, shows that the address 15E falls within GETINPUT. The displacement of the failing instruction, relative to the beginning of GETINPUT, is 15E – 110 = 4E.

Continued…

Module Map (cont'd)



```
0002E    44 LOOP    EQU   *
00002E 4150 0018          00018 45 LA   R5,24
000032 1B54                     46 SR   R5,R4
000034 4050 3000          00000 47 STH  R5,0(,R3)
000038 4110 C0C6          000CC 4B GET  IN,REC
000048 F273 C17A C127 00180 0012D 54 PACK DWD,RECTEMP
00004E 4F60 C17A          00180 55 CVB  R6,DWD
000052 9560 C126     0012C     56 CLI  RECSIGN,C'-'
000056 4770 CO5B     0005E     57 BNE  LOOPING
00005A 4C60 C182     00188     58 MH   R6,=H'-1'
```

Calculates entry number

Places in first half of array entry

Gets the next temp

Converts the absolute value to a fixed point

Is it negative?

No

Yes- change the sign

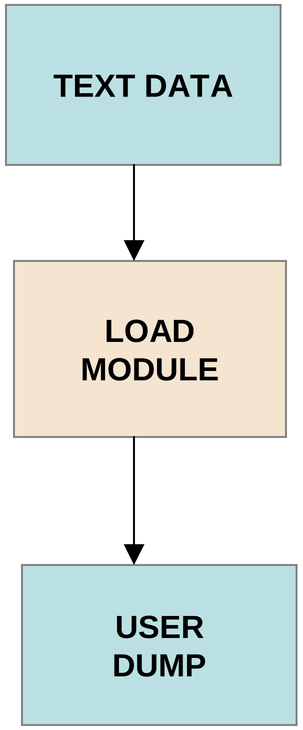
After viewing the module map, you then look at the listing of GETINPUT, to find what instruction failed. The relevant portion of the listing is shown in the example.

Batch Debugging

In some cases, it is possible that the symptom dump would not contain enough information to determine the root cause of the problem.

In such cases, the program can be re-run with a SYSUDUMP DD statement, in order to produce a full user dump.

User Dump:



JCL SPECIFIES //SYSUDUMP DO….

Continued…

Batch Debugging (cont'd)

If the program does run to completion, but produces incorrect results, the following methods for debugging can be applied:

- The code for logic errors should be checked and additional print statements should be included to display intermediate results.
- Areas of storage, or register contents or system control blocks should be dumped at specific points in the program. The system macro SNAP is used to produce a storage dump, and then continue execution. The system macro ABEND is used to produce a dump and terminate execution.

Debugging Techniques:

- **Check Program Logic**
- **Addition Print Statements**
- **Dump Registers, Storage and Continue (SNAP)**
- **Dump Registers, Storage and Terminate (ABEND)**

Interactive Debugging

A very powerful interactive debugging facility is provided by the IBM Interactive Debug Facility (IDF).

You compile and link the program with IDF included, and then run IDF in the TSO/E environment. IDF allows you to control the execution of your program, and interactively examine register and storage contents.

You can single step through your program, executing one instruction at a time. You can set break points, so that your program runs until it reaches a specific address, and then pause. You can monitor specified variables to ensure that only certain ranges of values are assigned.