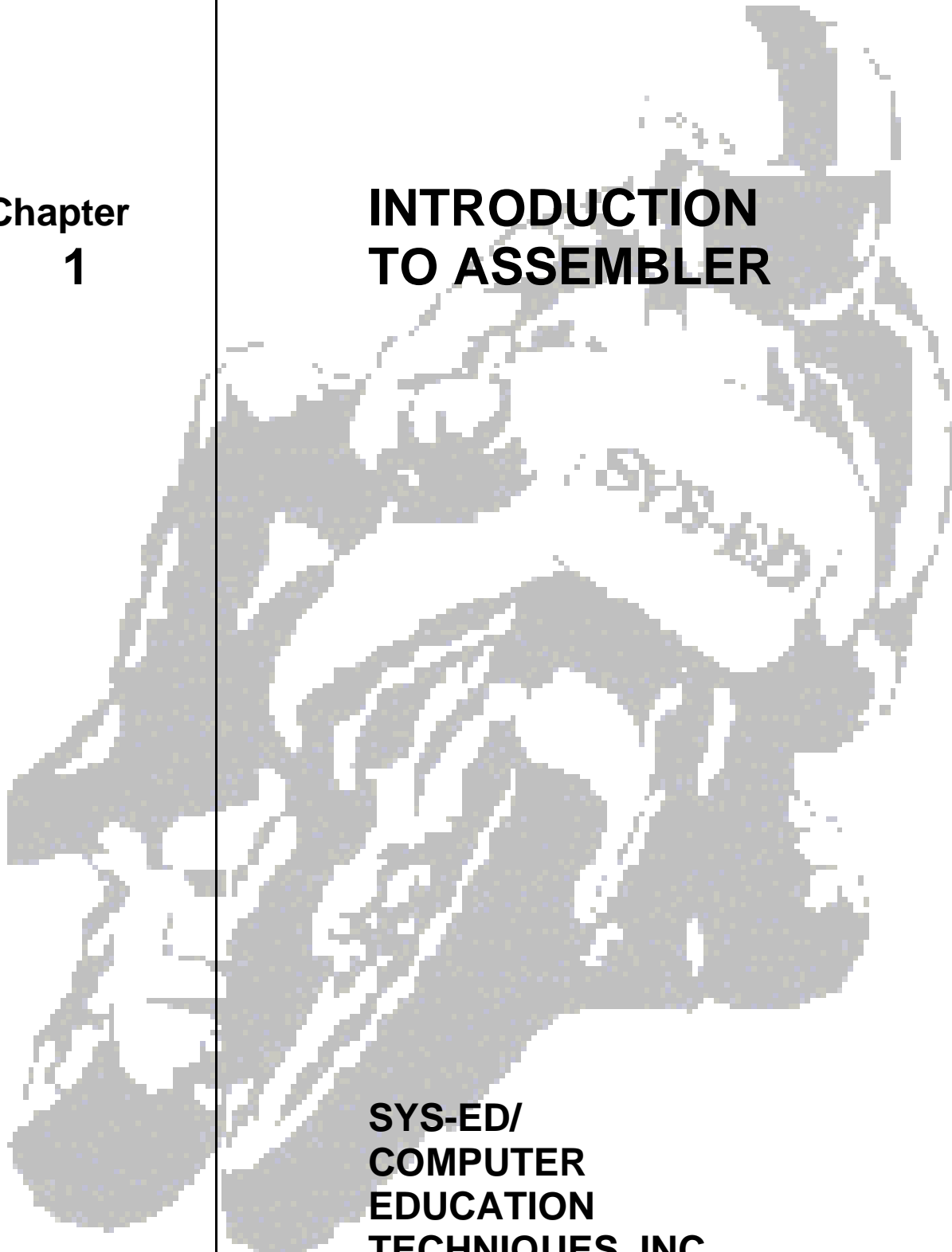


**Chapter
1**

**INTRODUCTION
TO ASSEMBLER**



**SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.**

Objectives

You will learn:

- C Structure of a ALC program.
- C Differences between assembler commands and macros.
- C The assembler and link process.
- C Structure of Macro and Copy libraries.
- C Concepts of DSECTs, symbols, addressability, relocatability, and addressing.
- C How to differentiate between machine instruction formats.

1 The Assembler Language

The function of the assembler is to translate programs written in the assembler language into object modules. The code will then be in a suitable form for processing by the Linkage Editor.

The assembler language is a symbolic representation of machine language. The correspondence between the two can be best understood by examining the listings produced by an assembler.

The assembler, like the linkage editor, is supplied as part of VSE, CMS, and MVS, and executes under control of the appropriate control program.

It accepts, as input, a source program written in the Assembler Language.

This consists of three types of instructions:

- C Machine instructions.
- C Assembler instructions.
- C Macro instructions.

As expected with an IBM language, code can also be copied in from a source statement library (VSE) or a pds (MVS) or MACLIB (CMS).

1.1. Machine Instructions

The machine instructions are translated into machine language code which can be executed. Most of the instructions in this course are machine instructions.

All machine instructions are listed in the System/370 Reference Summary, which is sometimes known as the "Yellow card" or "Yellow booklet".

Examples of machine instructions include all functional operations such as Move, Load, Add and Branch.

1.2. Assembler Instructions

The assembler instructions do not generally produce any object code. Rather, they issue instructions to the assembler itself to define constants, reserve storage areas, and define the start and end of a source module.

The following assembler instructions are typically included in a COBOL or PL/1 source listing:

EJECT Start a new page.

SPACE 3 Insert 3 blank lines in the source module listing.

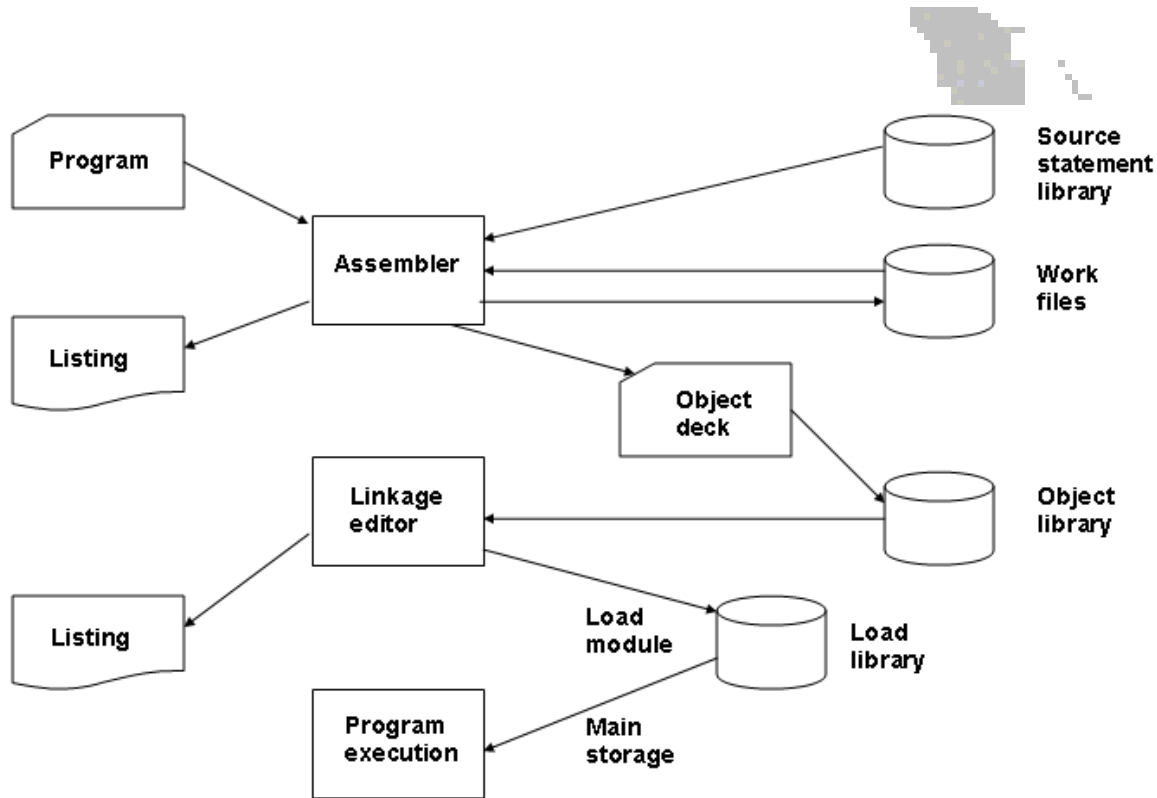
Further assembler instructions will be discussed later.

2 Macro Instructions

A macro is a pre-defined set of code. A macro instruction requests the assembler to process a named macro. As a result, the assembler generates machine and assembler instructions which are then processed as if they were part of the original source module.

Most macros you will use will be IBM-defined. It is also possible to prepare macro definitions and call them as appropriate.

3 Assembly and Link Editing



The output from the assembler includes the following:

Object module listings of:

- Source Module
- Object Code
- Diagnostics
- Cross-Reference Listings

4 Assembler Processing

The three types of assembler instructions are processed at different times as the following diagram shows:

Coding	Pre Assembly	Assembly	Link Edit	Program Fetch	Execution
Machine Instructions		-----			-----
Most Assembler Instructions		-----			
DC, DS, CCW		-----			-----
ENTRY, EXTRN, WXTRN Address Constants		-----	-----	-----	
PUNCH, REPRO		-----	-----		
Conditional Assembly & Macro Processing	-----				
MNOTES	-----				

5 Executing the Assembler

In VSE the assembler is loaded into main storage and starts to execute as a result of the statements:

```
// EXEC ASSEMBLY,SIZE=128K  
. source module  
/*
```

For OS, use:

```
//STEP EXEC ASMFCL  
//ASM.SYSIN DD *  
. source module  
/*
```

For CMS, use the CMS command:

```
ASSEMBLE prog
```

where the source module is in a file called

```
prog ASSEMBLE
```

6 Macro and Copy Libraries

The source module can include MACRO calls and COPY instructions which require access to libraries. The macro libraries contain IBM-supplied and user-written macro definitions. The copy library contains "books" of source code which may be incorporated into a program by COPY statements.

6.1. Control Sections

An assembler program is made up of one or more control sections or CSECT. A control section is the smallest relocatable unit of code. All elements of a control section are loaded into adjoining virtual storage locations. They contain code, storage, and constants.

6.2. DSECTS

You will also see DSECTS. These are Dummy Sections which do not contain executable code. Instead, they enable you to code a map of a storage area. You will see DSECTS used with table entries. DSECTS will be explained in detail in the Table Handling segment.

7 Symbols

Symbols are used in assembler programs to make the instructions easier to read. One of the chief advantages of using symbols is that they appear in a symbol cross-reference table which is printed in the program listings. Symbols can be used to represent registers, storage addresses, constants, operands and almost any element used in assembler.

The EQU (equat) instruction can be used to define symbols.

8 Data in Assembler Program

Assembler programs can process data in all the common formats.

Data can be defined in the following formats:

	COBOL	PL/1
Decimal	PIC 999	-----
Binary	PIC S999 COMP	FIXED BIN(x,0)
Packed Decimal	PIC S999 COMP-3	FIXED DEC(x,0)
Hexadecimal	-----	-----
Character	-----	-----

8.1. Addressability

8.2. Relocatability

Programs can be executed in any partition or region to the machine. They do not have to execute in a particular part of main storage, and are said to be **relocatable**.

A relocatable object module is one which can be loaded into any suitable virtual storage location without affecting program execution. As such, the module is said to be relocatable from its originally assigned storage area.

Before the days of virtual storage, modules would work only with a single set of addresses i.e. a version of a program could be executed only in a particular partition.

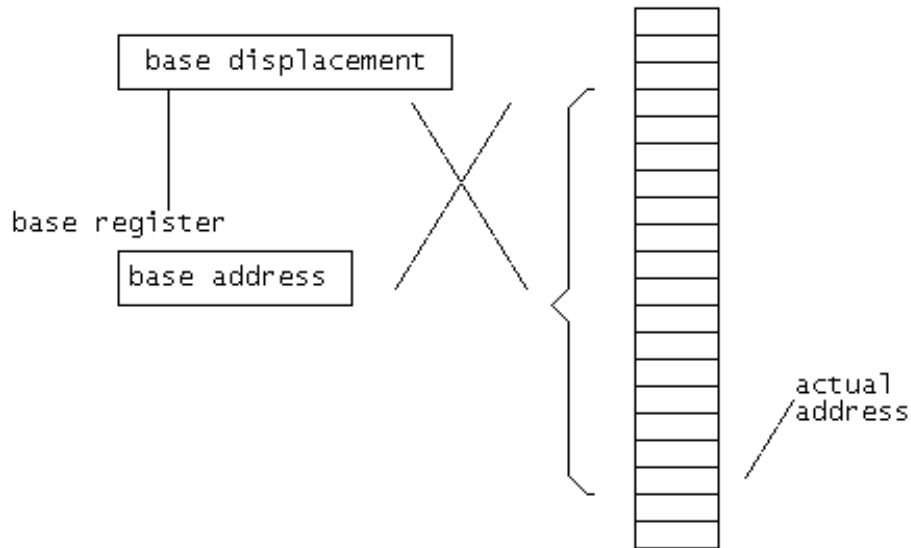
Relocability is helped by the fact that addresses are usually assembled in their base-displacement form.

9 Addressing

Addressing in assembler code are made up of two components:

base + displacement

The base is held in one of the general registers. The displacement is a number in the range 0 to 4095. This measures the distance in bytes from the base to the required location.



Thus if the **base** was held in Register 6 and the displacement was known to be 80 bytes, then the required address would be found by the addition:

```

Address held in Register   6
                        + 80
-----
required address
    
```

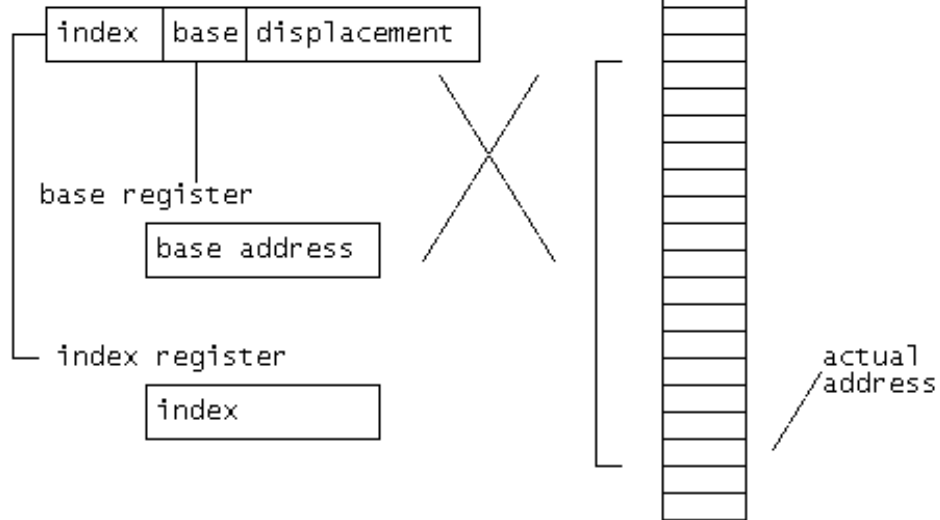
The value of Register 6 is not known at assemble time, but is determined at execution time. Thus if a program is executed in BG one day, and in F4 the next, it will have different values for the Register 6 component. Naturally, if the program uses identical data on each occasion, the contents of the target address will be the same each time.

A displacement of 80 can either be written as 80 or as X'50'.

Some locations may have addresses which are made up of :

base + displacement + index.

The base and the index components will be held in registers, and the displacement can be specified as a decimal or hex value.



In Assembler, you can choose which register will be your base register (subject to a few conventions). Register 12 is commonly chosen.

You may have noticed from the Register Assignment in COBOL listings how base registers are used for files and for working-storage (e.g. WORKING-STORAGE usually has a base register 6).

10 Registers

There are sixteen full word general purpose registers (GREGs) which can be used in your programs for holding addresses, counters, intermediate results, and so on. There are certain conventions which govern their usage.

Register 0	Is reserved for system use. Don't try to store any contents in it.
Register 1	Used by the system.
Register 13	Used by the system for module linkage.
Register 14	Return address.
Register 15	Branch to address.

This leaves you with 11 registers. This may seem enough, but they tend to get used up very quickly.

11 Establishing Addressability

Symbols can be used instead of addresses. This means that symbols can be used with meaningful names instead of explicit addresses in the base-displacement form.

If this is done, the assembler will convert the implicit addresses of the symbol into explicit addresses required by the machine instructions.

However, the assembler has to know:

- C A base address for evaluating addresses within a control section.
- C A base register to hold this address.

location counter

/

0 SAMPLE START 0 0 is the default
 SAVE (14,12) save the registers -
 OS and subroutines

4 BALR 12,0 initialise a base register
 USING *,12 tell Assembler which one

6 FIRST LA 10,ADDRESS 41 A 0 C05E
 (= code generated)

NORMAL
WAY OF
CODING

or:

LA 10,X`5E`(0,12) 41 A 0 C05E
 (same as above)

POSSIBLE
WAY OF
CODING

64 ADDRESS DC C`EXAMPLE' base

RETURN (14,12) restore the registers
 END SAMPLE

NOTE:

The SAVE macro is 4 bytes long, and the BALR instruction is 2 bytes long, which is why the location counter has a value of 6 at symbol FIRST.

The Assembler starts counting its addresses from this point, so the displacement of ADDRESS from the start of the program is:

$$x`64' \text{ minus } x`6' = x`5E'$$

12 Saving Registers

When an OS assembler program starts, or when any program is CALLED, the first action taken is to SAVE the contents of the registers (i.e. used by the previous or calling program).

These values are restored at the end of the program by a RETURN (for OS) or EOJ (for VSE) macro. They will be discussed in more detail later in the course.

The BALR instruction loads Register 12 (the base register) with the base address. It is this instruction which establishes addressability.

At execution time, Register 12 will be loaded with the appropriate address; all subsequent instructions which reference Register 12 will be able to access the correct value for the base address.

13 Coding Conventions

Statements may start in column 1 and end in column 71. If continuation is required, a non-blank character (e.g. X) must be placed in column 72, and the subsequent line must start in column 16.

Columns 73 to 80 can be used for sequence number and/or identification characters.

The most important columns are 1 to 71:

col 1 to 8	Contains a label.
col 10 to 71	Contains operation, operands, comments.

The operations (e.g. LA) starts in column 10, although assembler can be coded in free form.

The operands start in column 16 and are separated from each other by commas.

Comments are optional. They are separated from the operands by one or more blanks. Many programmers start comments in column 31.

ALWAYS use comments to document your coding, otherwise it will be difficult to read and maintain. Some of the IBM supplied assembler is particularly well documented.

```
LABEL LA    R10,FRED          LOAD ADDRESS OF SYMBOL FRED
*                                     INTO REGISTER 10
.
.
.
FRED  DC    C'HELLO'
```

Notice that a complete line can be comments if it begins with an asterisk (*).

14 Operating System Differences

Assembler will run in CMS, VSE or MVS. The VSE assembler is a subset of the OS (used in MVS and CMS) assembler, although the differences are small. Object code produced is compatible across systems.

Each system - MVS, CMS, VSE - uses its own set of macros, and these are not compatible. Therefore, a program written for execution under VSE will not run under MVS without recoding of the macros.

In VM/VSE systems, the VSE assembler will not run under CMS. However, programs assembled under CMS are object code compatible provided VSE macros are used. The systems programmer will have to set up a CMS MACLIB containing VSE macros.

15 Machine Instruction Formats

There are six formats for the machine instructions discussed on this course. The formats describe how Registers and storage addresses (explicit or implied) can be used to specify operands to and instruction.

The formats are:

RR	R1,R2	register register
RX	R1,D2(X2,B2)	register index
RS	R1,R3,D2,(B2) or R1,M3,D2,(B2)	register storage
SI	D1,(B1),I2	storage immediate
SS	D1,(L1,B1),D2(L2,B2)	storage storage
SS	D1(L,B1),D2(B2)	storage storage

Displacements are in decimal unless you code X`..'

16 Receiving Field in an Operation

In general, the second operand is moved, added, loaded etc., into the first operand. In other words, the transfer of data is from the second operand to the first.

Exceptions are the **STORE** instructions and the **Convert to Decimal** instructions.



17 RR Format

Instructions in which both operands are registers are coded in RR format.

The machine code format for RR instruction is:

Op	R1	R2
0	7	15

The corresponding Assembler format is:

```
Op R1,R2
```

The RR format instruction Add Register has a mnemonic Op code AR in Assembler and the corresponding machine code is X'1A' therefore;

```
AR 1,11
```

```
assembles to 1A1B
```

Since the registers are identified by their number, it is convenient to use a suitable symbol in Assembler language to avoid confusion.

The common symbols are:

```
R0 for register 0  
R1 for register 1  
.  
.  
.  
R15 for register 15
```

The symbols can be set up by the EQU (equates) instruction.

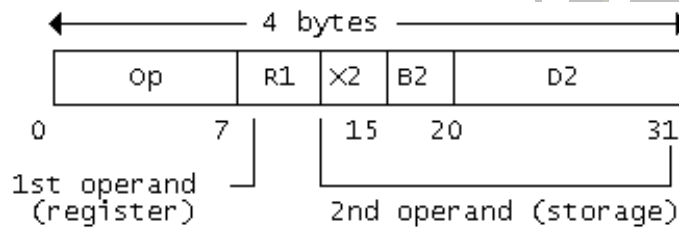
Therefore, the example could have been coded as:

```
AR R1,R11 MORE READABLE
```

18 RX Format

Register index instructions have their first operand in a register and their second in storage. The latter is addressed by base, displacement and index.

The machine code format for RX instructions is:



The Assembler format is:

```
A R1,D2(X2,B2)
```

The second operand could use base, index, and displacement components to explicitly define a location.

For example:

```
Register 8 contains 00 00 00 14
Register 9 contains 00 00 10 00
Register 10 contains 00 00 24 00
```

4 bytes at Location 3404 contain 00 00 01 A3

The instruction:

```
A 8,4(9,10)
```

or

```
A R8,4(R9,R10)
```

would cause register 8 to contain 00 00 01 B7.

19 Normal Way of Coding

In practice, the location at 3404 would be defined with a symbolic name although in some cases RX instructions are coded with displacement, base and index given explicitly. This means that the symbolic name can be used instead of the D2(X2,B2) operand, since it implies an address.

Examples:

```
A R3,NUM
A R3,NUM(R4)
A R3,NUM+4
A R3,*
LA R3,24(R3,R4)
LA R3,24(R3,0)
LA R3,24(R3)
LA R3,24
```

If the base, index and displacement is specified explicitly these will be used in the machine code translation.

If a symbol is used for specifying an address, then the Assembler will generate a base and displacement from the information it has been given in the USING instruction and the index will be zero unless explicitly specified.

Examples:

Assembler	Machine code
A R3,24(R4,R6)	5A34601 8
A R3,NUM	5A30C04 E
A R3,NUM(R4)	5A34C04 E

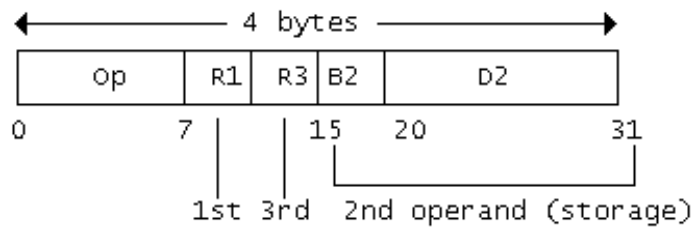
Note:

The last two examples assume USING *,12 and the displacement would be calculated by the Assembler.

20 RS Example

The RS format uses three operands the second of these is in storage and uses base and displacement addressing.

The machine code format is:



The Assembler format is:

Op R1,R3,D2(B2)
Op R1,R3,Symbol

Note: There will be a R3 or M3 depending on the context of the instruction.

An example of this format is the LM (Load Multiple) instruction. This loads a number of registers with the contents of the same number of consecutive fields at a specified location. The first and third operands are used together to indicate which registers are to be loaded.

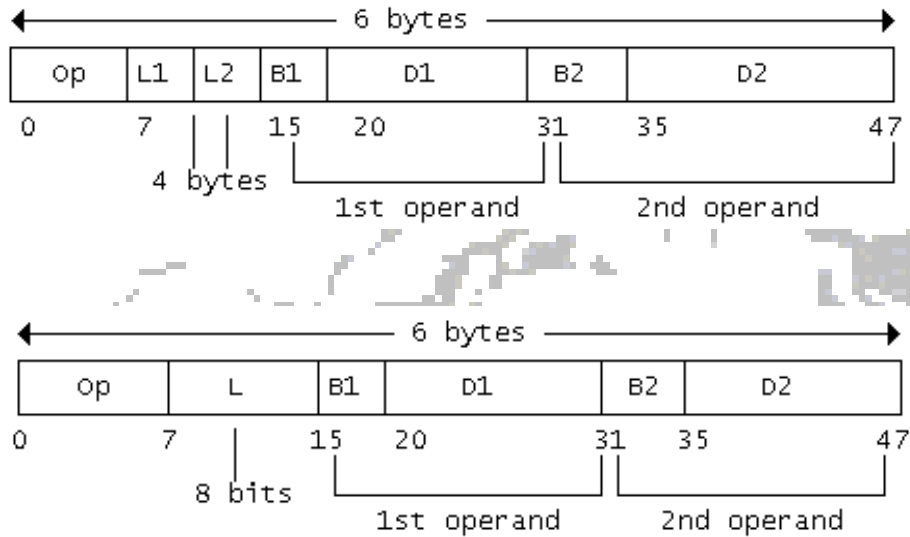
Examples:

Assembler	Machine code
LM R3,R7,24,(R10)	9837A018
LM R3,R7,NUM	9837C04E
LM R3,R7,NUM+4	9837C052

21 SS Format

There are two types of SS (storage-storage) formats; with one format, the two fields have the same length; with the other, the lengths are different.

The machine code formats are:



The Assembler formats are:

- Op D1(L1,B1),D2(L2,B2)
- Op Symbol1(L1),Symbol2(L2)
- Op D1(L,B1),D2(B2)
- Op Symbol1(L),Symbol2
- Op Symbol1,Symbol2

Examples:

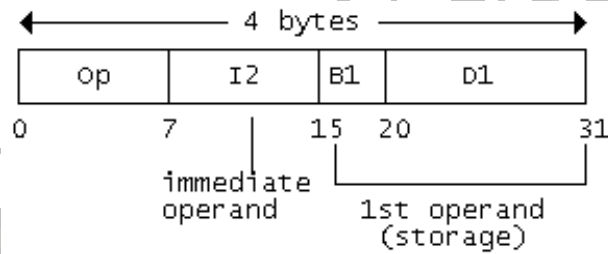
- MVC TO,FROM
- MVC TO(5),FROM
- MVC TO+12,FROM
- MVC TO+12(7),FROM
- MVC TO+12(L,FROM),FROM
- MVC NAME(L,NAME1),NAME2

22 SI Format

The Storage Immediate format uses a single byte coded within the instruction itself to hold the data for the second operand.

SI format cannot be used with variable data in the second operand it can only be used when the data is the same whenever the program is executed.

The machine code format is:



The Assembler format is:

```
Op D1(B1),I2
Op Symbol,I2
```

Examples:

Assembler	Machine code
MVI NUM,27	921BC04E
MVI INITIAL,C'J'	92D1C040
MVI NUM,X'4A'	924AC04E

23 Long Control Sections

The USING Instructions has effect for 4096 bytes.

If a control section is longer than 4096 bytes, more than one base register must be assigned. In the following example, registers 10,11,12 are used:

```

        .
        .
        .
        BALR 10,0
        USING FIRST,10,11,12
FIRST   LM    11,12,NEXT
        B     BEGIN
NEXT    DC    A(FIRST+4096,FIRST+8192)
10 BEGIN DS   OH
        .
        .
        .
        FIRST+4095
        FIRST+4096
11
        FIRST+8192
12
    
```

Alternative:

```

        BALR, 10,0
        USING *,10,11,12
        LA 11,2048(10)   R11 = R10 + 4096
        LA 12,1
        LA 12,4095(11,12) R12 = R11 + 4096
    
```