Chapter
1

ASSEMBLER LANGUAGE: INTRODUCTION

TM

**SYS-ED/
Computer
Education
Techniques, Inc.**

Objectives

You will learn:

- Differences between machine and assembler instructions.
- Macro instructions.
- Assembly and link editing.
- Assembler processing.
- Macro and copy libraries.
- Control Sections.
- Assembler concepts - addressability, relocability, and registers.
- Coding conventions.
- Machine instruction formats - RR, RX, RS, SS, and SI.
- How to work with long control sections.

1 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 the operating system, 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:

• Machine	• Assembler	• Macro

As expected with an IBM language, code is stored in a PDS: Partitioned Dataset with a record length of 80. Macros also are stored in a PDS referred to as the macro library or MACLIB.

1.1 Machine Instructions

The machine instructions are translated into machine language code which can be executed.

Examples of machine instructions include 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.

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

EJECT Start a new page.

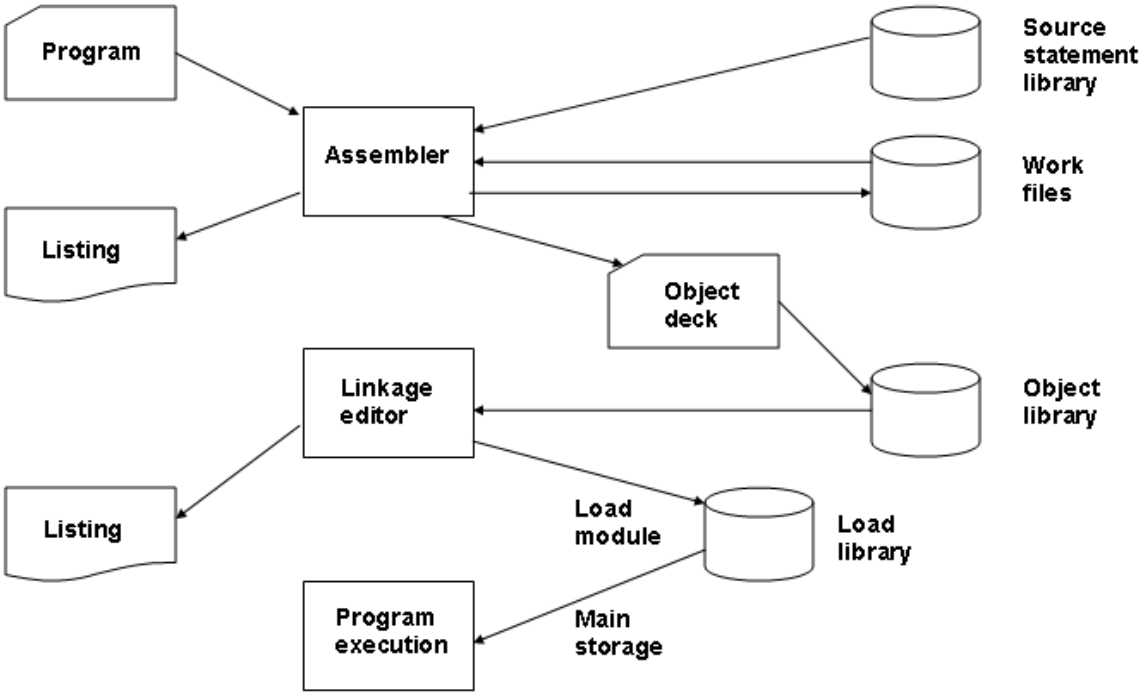
SPACE 3 Insert 3 blank lines in the source module listing.

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 will be IBM-defined. It also is possible to prepare macro definitions and call them when required.

3 Assembly and Link Editing



The output from the assembler includes the object module listings:

- Source module
- Object code
- Diagnostics
- Cross-reference listings

4 Assembler Processing

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 and Macro Processing	----------				
MNOTES	----------				

5 Executing the Assembler

An installation will have JCL or an environment such as Endevor or ChangeMan to assemble a program.

The standard JCL supplied by IBM will assemble and link an Assembler program.

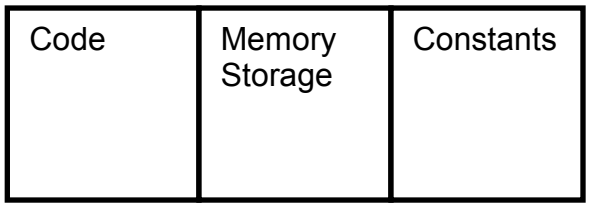
```
//STEP      EXEC  ASMACL
//ASM.SYSIN  DD   *
.
.   source module
.
/*
```

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

DSECTs are Dummy Sections which do not contain executable code. Instead, they provide the capability to code a map of a storage area.

A DSECT or Dummy Section provides the programmer with the ability to describe the layout of an area of storage without reserving virtual storage for the area that is described. The DSECT layout can then be used to reference any area of storage which is addressable by the program.

A DSECT is a form of a template or pattern, which can be used on any area of storage. Once the DSECT is positioned, the symbolic names in the DSECT can be used to extract data from the underlying storage area.

7 Symbols

Symbols are used in assembler programs to make the instructions easier to read.

A major advantage associated with 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 programs.

The EQU - equate instruction can be used to define symbols.

8 Data in Assembler Program

Assembler programs can process data in all the common formats.

Data is defined with a DS - Data Storage or DC- Data Constant command.

Data can be defined in the following formats:

	COBOL	**PL/1**
Decimal	PIC 999	PIC '9999'
Binary	PIC S999 COMP	FIXED BIN(x,0)
Packed Decimal	PIC S999 COMP-3	FIXED DEC(x,0)
Hexadecimal	----------	----------
Character	PIC X(10)	CHAR(x)

9 Addressability

9.1 Relocability

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 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; this meant that 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.

10 Addressing

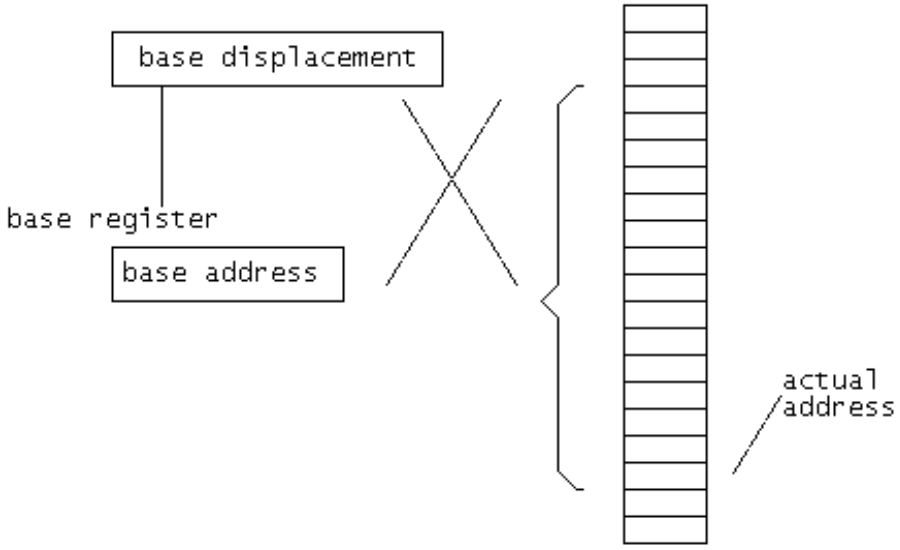
Addressing in assembler code is made up of two components:

base + displacement

A specific byte is indicated by specifying a beginning address, known as the base and an offset from the base, known as the 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.



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.

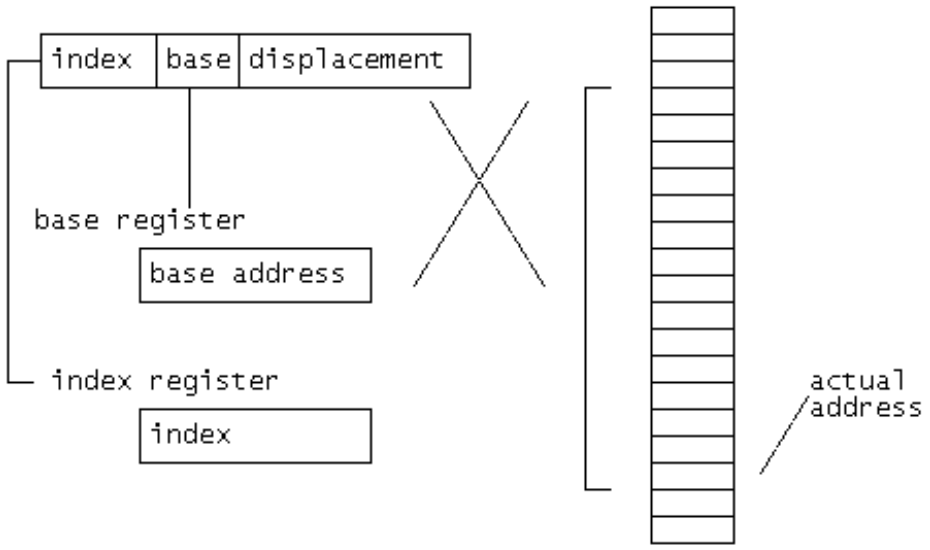
If a program is executed one day, and again the next day, it will have different values for the Register 6 component. Naturally, if the program uses identical data on each occasion, the contents of the target address will be the same each time.

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; the displacement can be specified as a decimal or hex value.



With the Assembler language, subject to a few conventions, there is the ability to choose which register will be the base register. Register 12 is the common choice.

11 Registers

There are sixteen full word GREGs: General Purpose Registers which can be used in programs for holding addresses, counters, intermediate results, and so on.

There are certain conventions which govern the usage.

Register 0	Is reserved for system use. Do not attempt to store any content 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 11 registers. Although this may seem sufficient, they tend to get used up very quickly.

12 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.

The assembler will need to have:

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

13 Location Counter

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

4           BALR    12,0               Initialize a base register.
            USING   *,12               Inform assembler which one.

6  FIRST    LA 10,ADDRESS              41 A 0 C05E          NORMAL
                                       (= code generated)   WAY OF CODING

       or:

            LA 10,X`5E`(0,12)          41 A 0 C05E
                                                            POSSIBLE
                                       (Same as above.)     WAY OF CODING

64 ADDRESS  DC  C`EXAMPLE'             Base.

            RETURN  (14,12)            Restore the registers.
            END  SAMPLE
```

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

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

x`64' minus x`6' = x`5E'

14 Saving Registers

When a z/OS Assembler language program starts, or when any program is CALLED, the first action taken is to SAVE the contents of the registers used by the previous or calling program.

These values are restored at the end of the program by a RETURN for z/OS.

The BALR instruction loads Register 12, which is 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 instruction which references Register 12 will be able to access the correct value for the base address.

15 Coding Conventions

Statements may start in column 1 and end in column 71. If continuation is required, a non-blank character, such an 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 identification characters.

The most important columns are 1 to 71:

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

The operations, such as 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.

It is important to use comments for documenting coding; otherwise it will be difficult to read and maintain.

Example: Comment

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

A complete line can be comments if it begins with an asterisk (*).

16 Machine Instruction Formats

The formats for machine instructions describe how registers and explicit or implied storage addresses can be used to specify operands to and instruction.

The formats to be examined 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 to storage
SS	D1(L,B1),D2(B2)	storage to storage

Displacements are in decimal unless coded with X`..'.

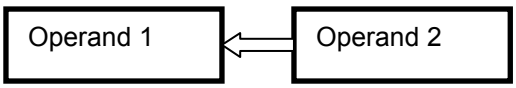
16.1 Receiving Field in an Operation

In general, the second operand is moved, added, loaded etc., into the first operand.

This means that the transfer of data is from the second operand to the first.

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

Transfer of Data



17 RR Format

The RR format is an instruction in which both operands are registers coded in RR format.

The machine code format for RR instruction is:

Op		R1		R2	
0	7	8	11	12	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
```

which assembles to 1A1B

Since the registers are identified by 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
```

Many programmers will code the following equates into every program:

```
R0      EQU   0
R1      EQU   1
R2      EQU   2
R3      EQU   3
R4      EQU   4
R5      EQU   5
R6      EQU   6
R7      EQU   7
R8      EQU   8
R9      EQU   9
R10     EQU   10
R11     EQU   11
R12     EQU   12
R13     EQU   13
R14     EQU   14
R15     EQU   15
```

18 RX Format

Register index format 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:

Op		R1		X2		B2		D2	
0	7	8	11	12	15	16	19	20	31

The assembler format is:

A R1,D2(X2,B2)

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

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 3414 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.

18.1 RX: Format - Normal Way of Coding

In practice, the location at 3414 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)	5A34CO4 E

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

19 RS Format

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

The machine code format is:

Op		R1		R3		B2		D2	
0	7	8	11	12	15	16	19	20	31

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.

The LM - Load Multiple instruction is an example of this format. This loads a number of a 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

20 SS Format

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

Two of the machine code formats are:

Op		L1		L2		B1		D1		B2		D2	
0	7	8	11	12	15	16	19	20	31	32	35	36	47

Op		L1		B1		D1		B2		D2	
0	7	8	15	16	19	20	31	32	35	36	47

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
```

21 SI Format

The Storage Immediate format instruction 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:

Op		I2		B1		D1	
0	7	8	11	16	19	20	31

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

22 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.

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
```