Assembling, Linking and Executing

1) Assembling:

- Assembling converts source program into object program if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- A header is created which contains the incomplete address in front of the generated **obj** module during the assembling.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.
- For short programs, assembling can be done manually where the programmer translates each mnemonic into the machine language using lookup table.
- Assembler reads each assembly instruction of a program as ASCII character and translates them into respective machine code.

Assembler Types:

There are two types of assemblers:

a) One pass assembler:

- This assembler scans the assembly language program once and converts to object code at the same time.

- This assembler has the program of defining forward references only.
- The jump instruction uses an address that appears later in the program during scan, for that case the programmer defines such addresses after the program is assembled.

b) Two pass assembler

- This type of assembler scans the assembly language twice.
- First pass generates symbol table of names and labels used in the program and calculates their relative address.
- This table can be seen at the end of the list file and here user need not define anything.
- Second pass uses the table constructed in first pass and completes the object code creation.
- This assembler is more efficient and easier than earlier.

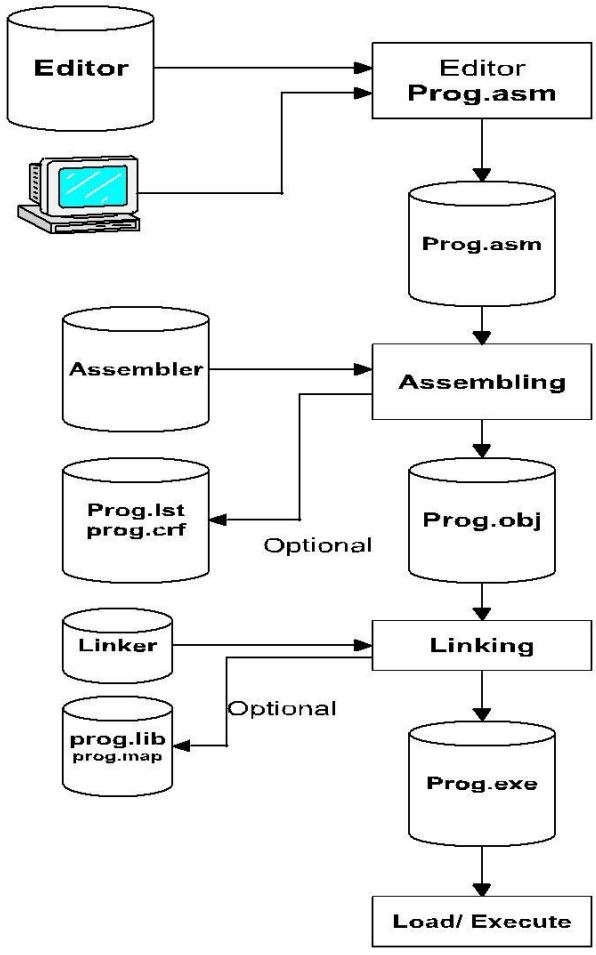

Fig: Steps in assembling, linking & Executing

2) Linking:

- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.
- It completes the address left by the assembler.

- It combines separately assembled object files.
- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3) Loading and Executing:

- It Loads the program in memory for execution.
- It resolves remaining address.
- This process creates the program segment prefix (PSP) before loading.
- It executes to generate the result.

Sample program → assembling → object Program → linking → executable program

Writing .COM programs:

- It fits for memory resident programs.
- Code size limited to 64K.
- **.com** combines PSP, CS, DS in the same segment
- SP is kept at the end of the segment (FFFF), if 64k is not enough, DOS Places stack at the end of the memory.
- The advantage of **.com** program is that they are smaller than **.exe** program.
- A program written as **.com** requires ORG 100H immediately following the code segment's SEGMENT statement. The statement sets the offset address to the beginning of execution following the PSP.

```
.MODEL TINY
.CODE
ORG 100H                    ; start at end of PSP
BEGIN:JMP MAIN              ;Jump Past data
        VAL1 DW 5491
        VAL2 DW 372
        SUM DW ?
MAIN: PROC NEAR
        MOV Ax, VALL
        ADD AX, VAL2
        MOV SUM, AX
        MOV AX, 4C00H
        INT 21H
MAIN ENDP
END BEGIN
```

Macro Assembler:

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name.

- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
- Macro are useful for the following purposes:
  - To simplify and reduce the amount of repetitive coding.
  - To reduce errors caused by repetitive coding.
  - To make an assembly language program more readable.
  - Macro executes faster because there is no need to call and return.
  - Basic format of macro definition:

```
Macro name      MACRO [Parameter list]      ; Define macro
                ……
                ……
                [Instructions]              ; Macro body
                ……
                ……
                ENDM                        ; End of macro
```

E.g. **Addition**

```
MACRO
IN AX, PORT
ADD AX, BX
OUT PORT, AX
ENDM
```

Passing argument to MACRO:

- To make a macro more flexible, we can define parameters as dummy argument

```
Addition MACRO VALL1, VAL2
        MOV  AX, VAL1
        ADD AX, VAL2
        MOV SUM, AX
        ENDM

.MODEL SMALL
.STACK 64
.DATA
        VAL1  DW   3241
        VAL2 DW 571
        SUM DW ?
.CODE
MAIN PROC FAR
        MOV AX, @ DATA
        MOV DS, AX
```

```
        Addition VAL1, VAL 2
        MOV AX, 4C00H
        INT 21H
MAIN ENDP
END MAIN
```

Addressing modes in 8086:

Addressing modes describe types of operands and the way in which they are accessed for executing an instruction. An operand address provides source of data for an instruction to process an instruction to process. An instruction may have from zero to two operands. For two operands first is destination and second is source operand. The basic modes of addressing are register, immediate and memory which are described below.

1) **Register Addressing:**

   For this mode, a register may contain source operand, destination operand or both.

   E.g. MOV AH, BL

   MOV DX, CX

2) **Immediate Addressing**

   In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes. This mode contains a constant value or an expression.

   E.g. MOV AH, 35H

   MOV BX, 7A25H

3) **Direct memory addressing:**

   In this type of addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it. One of the operand is the direct memory and other operand is the register.

   E.g. ADD AX, [5000H]

   Note: Here data resides in a memory location in the data segment, whose effective address may be computed using 5000H as the Offset address and content of DS as segment address. The effective address, here, is **10H*DS + 5000H**.

4) **Direct offset addressing**

   In this addressing, a variation of direct addressing uses arithmetic operators to modify an address.

   E.g. ARR DB 15, 17, 18, 21

   MOV AL, ARR [2] ; MOV AL, 18

   ADD BH, ARR+3 ; ADD BH, 21

5) **Indirect memory addressing:**

   Indirect addressing takes advantage of computer's capability for segment**:** offset addressing. The registers used for this purpose are base register (BX and BP) and index register (DI and SI)

   E.g. MOV [BX], AL

ADD CX, [SI]

6) Base displacement addressing:

This addressing mode also uses base registers (BX and BP) and index register (SI and DI), but combined with a displacement (a number or offset value) to form an effective address.

E.g. MOV BX, OFFSET ARR

≡ LEA BX, ARR

MOV AL, [BX +2]

ADD TBL [BX], CL

TBL [BX] ⇨ [BX + TBL] e.g. [BX + 4]

7) Base index addressing:

This addressing mode combines a base registers (BX or BP) with an index register (SI or DI) to form an effective address.

E.g. MOV AX, [BX +SI]

ADD [BX+DI], CL

8) Base index with displacement addressing

This addressing mode, a variation on base- index combines a base register, an index register, and a displacement to form an effective address.

E.g. MOV AL, [Bx+SI+2] ADD

TBL [BX +SI], CH

9) String addressing:

This mode uses index registers, where SI is used to point to the first byte or word of the source string and DI is used to point to the first byte or word of the destination string, when string instruction is executed. The SI or DI is automatically incremented or decremented to point to the next byte or word depending on the direction flag (DF).

E.g. MOVS, MOVSB, MOVSW

Examples:

```
TITLE Program to add ten numbers
        .MODEL SMALL
        .STACK 64
        .DATA
ARR DB 73, 91, 12, 15, 79, 94, 55,
89 SUM DW ?
        .CODE
MAIN PROC FAR
    MOV AX, @DATA
    MOV DS, AX
```

```
    MOV CX, 10
    MOV AX, 0
    LEA BX, ARR
L2: ADD Al, [BX]
    JNC L1
    INC AH
L1: INC BX
    LOOP L2
    MOV SUM, AX
    MOV AX, 4C00H
    INT 21H
MAIN ENDP
END MAIN
```