CHAPTER 13

Assembly Language Programming

13.1 Introduction to Assembly Language

Microprocessor chip designers create a basic set of instructions for every processor they design. These are really simplistic instructions that take baby steps as compared with high-level languages such as C++.

- Each instruction has a code so that the instruction decoder can decode them in order to energize the proper circuits to execute the function using the registers of the processor. These are referred to as ***operation code*** (OPCODE). *"Machine code is the instruction set that machine understands"*.
- Machine language is the only language understood directly by the CPU in our computers.
- Humans, however, understand words, so each machine code is given an "English" equivalent. *These instructions in text form are called **Mnemonic***.
- Assembly language is a mnemonic representation of machine code.
- Assembly language is not just a simple mapping of numbers to words. It also contains many high-level-language type constructs to make data definition and program structuring easier.
- There is a one-to-one correlation between assembly language instructions and the machine code.

Reasons for learning the assembly language

The assembly language is learnt for the following reasons :

(1) It helps to learn about the internal architecture of the computer. The more you use the assembly language the more knowledge you gain about the architecture.

(2) So as to make use of the utilities of a computer e.g. to directly communicate with the microprocessor of the computer. This is useful under the circumstances where programming in high level language is difficult or even impossible.

(3) Most of the times a high level language program written for the above applications does not yield the required performance. So use of assembly level programming is always a better solution.

(4) The high level language programs may need a large memory space. Instead the memory requirement will reduce if we use the assembly language programming.

There are less number of restrictions or rules as compared to the high level languages.

1.1 Programming Methodology

Generally the programmers do not write large programs using the assembly language.

Instead they write short, specific routines and subroutines in the assembly language.

Write the main long program using the high level language and call the smaller subroutines written in assembly language as and when required.

Assembly language subroutines can be written to handle operations which are not available in the higher level language.

13.2 Development of an Assembly Language Program

Tools used for development

The development of assembly language program needs following tools :

- Assembler
- Linker
- Loader
- Debugger and
- Emulator

Some of these tools are used for program development, some for the program execution and the remaining are useful for testing the assembly language program.

13.2.1 Steps for Developing an Assembly Language Program

- Developing an assembly language program is a four step process. The steps are as follows :
  - (i) To specify the source code as per the assembly language definition.
  - (ii) Assemble the program to create the object code.
  - (iii) Link the program to create an executable code.
  - (iv) Test and debug the program.

Fig. 13.2.1 shows the steps involved in developing and executing an assembly language program.

Description

- Refer Fig. 13.2.1 to understand the program development steps.

Step 1 :

- The first step is to analyse what the program is to do and how we want the program to do it.
- Then using an editor create the source file for program.

Step 2 :

- The second step is to assemble this source file.
- If the assembler indicates errors then use the editor for correcting them and again assemble this source file.

12 V

Step 3 :

- If the program consists of several modules, then use the linker to join them into one large object module.
- If the system needs to locate a program in order to specify its location in the memory then use the locator.
- At this stage the program is ready for loading into the memory and run.

Step 4 :

- If the developed program does not interact with any external hardware other than that connected directly to the system then use debugger for running and debugging your program.
- If the program is supposed to work with the external hardware system such a microprocessor based instrument then use emulator to run and debug the program.

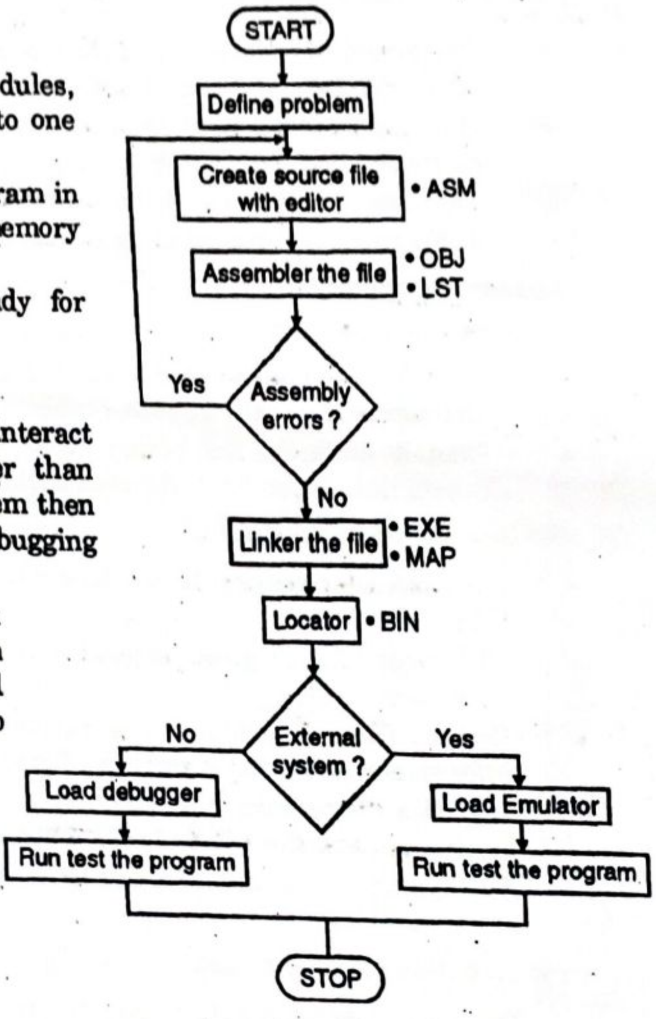

Fig. 13.2.1

13.3 Assembly Language Program Development Tools

- In this section we will go into the details of some of the assembly language program development tools.

13.3.1 Editor

- An editor is basically a software (i.e. a program).
- It helps the user to create a file that contains the assembly language statements.
- The examples of editors used for the assembly language programs are Wordstar, Edit, WordPad, Notepad etc.
- The job of the editor is to store the ASCII codes for the letters and numbers in the successive RAM locations.
- As the typing of program is over, this file is stored on a floppy or hard disk.
- This file is called as the "source file" and an ASM extension is given to it.
- The source file is then processed using an assembler.

13.4.2 Program Format and Assembler Directives

- First let us see the structure of general assembler program for 80813.
- The general assembler program structure for 8086 is shown in Figs. 13.4.2 (a) and 13.4.2 (b).

```
Line 1    .MODEL SMALL          ; Selects small model.
Line 2    .DATA                 ; Indicates data segment.
          ......      }
             ......   }         ; Data declaration
                                ; Indicates code segment.
Line 15   . Code
          ......      }
                      }         ; Program body.
          ......      }
          ......
Line n         End              ; End of file.
```

Fig. 13.4.2(a) : Program structure

```
DATA_HERE SEGMENT
.............................. }
                               }  ; Data declaration.
......
DATA_HERE ENDS
CODE_HERE SEGMENT
   ASSUME CS : CODE_HERE, DS : DATA_HERE
......                          }
......                          }  ; Body of the program
......
CODE_HERE ENDS
END
```

Fig. 13.4.2(b) : Program structure

13.5 Assembler Directives

Q. What are different types of assembler directives? Explain any four assembler directives.

Assembly language consists of two type of statements viz.

1) Executable statements

   These are the statements to be executed by the processor. It consists of the enti instruction set of 8086 (as seen in the chapter 3.)

2) Assembler directives

   These are the statements that direct the assembler to do something. As the na says, it direct the assembler to do a task.

- The speciality of these statements is that they are effective only during the assem of a program but they do not generate any code that is machine executable.
- We can divide the assembler directives into two categories namely the gen purpose directives and the special directives.

They are classified into the following categories based on the functions performed by them.

73

• Simplified segment directives	• Data allocation directives
• Segment directives	• Macros related directives
• Code label directives	• Scope directives
• Listing control directives	• Miscellaneous directives

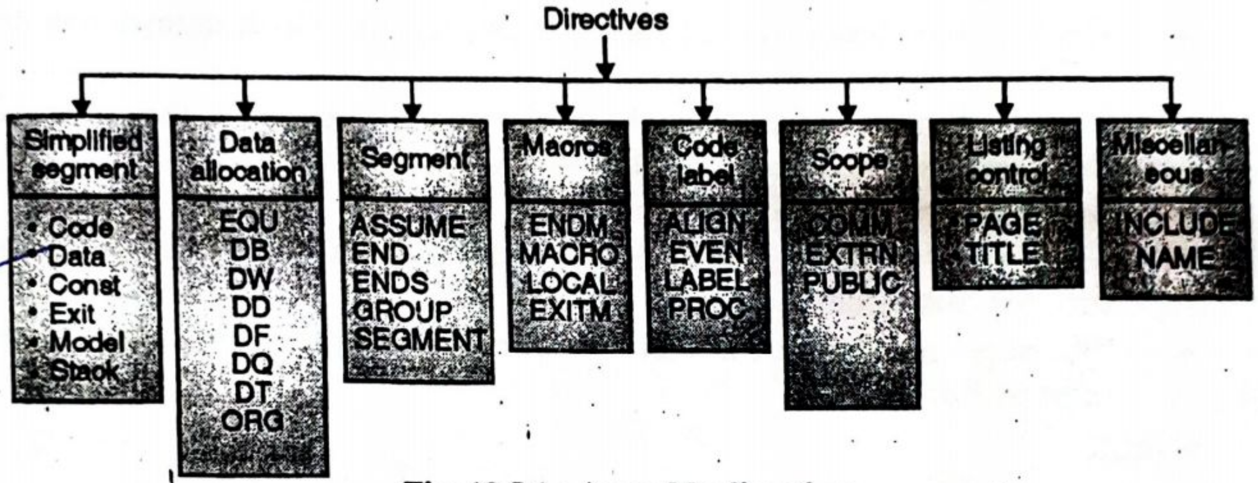

Fig. 13.5.1 : Assembly directives

1. . CODE

- This assembler directive indicates the beginning of the code segment. Its format is as follows :

  . CODE [name]

- The name in this format is optional.
- For tiny, small and compact models the segment name is – TEXT always.
- The medium and large memory models use more than one code segments which can be distinguished by name.

2. . DATA

This directive indicates the beginning of the data segment.

3. .MODEL

- This directive is used for selecting a standard memory model for the assembly language program.
- Each memory model has various limitation depending on the maximum space available for code and data.
- The general format for defining the Model directive is as follows :

  . MODEL [memory model] :

- The memory model is chosen based on the user's requirement by referring to Table 13.5.1.

Table 13.5.1

Model	Number of Code Segments	Number of Data Segments
Small	One code segment and of size < = 64 Kbytes	One of size < = 64 Kbytes
Medium	Code segment may be of any number and any size.	One of size < = 64 Kbytes

Model	Number of Code Segments	Number of Data Segments
Compact	One of size < = 64 Kbytes	Data segment of any number and any size.
Large	Any number, any size	Any number, any size.
Huge	Any number, any size	Any number, any size.

- The size of a memory model can be anything from small to huge.
- The tiny model is meant for the . COM programs because they have their code, data and stack in only one 64 kB segment of memory.
- On the other hand the flat model is the biggest which defines one area upto 4 GB for code and data.
- The small model is useful for the student level programs because for this model the assembler assumes that the addresses are within a span of 64 kB and hence generates 16 kB offset addresses.
- In the compact model, the assembler can use 32 bit addresses. So the execution time for this model is longer.
- The huge model contains variables such as arrays which need a larger space than 64 kB.

4. STACK

- This directive is used for defining the stack. Its format is as follows :

  . STACK [size]
- The size of the stack is 1024 bytes by default but this size can be overridden.
- We can omit the . stack command if the stack segment is not to be used in a program.
- Example of the stack directive is

  . STACK 100

Which reserves 100 bytes for the stack segment.

5. EQU-Equate

- It is used to give a name to some value or symbol in the program.
- Each time when the assembler finds that name in the program, it replaces that name with the value assigned to that r variable .
- Its format is [name] EQU initial value.

  e.g. : FACTORIAL EQU 05H.
- This statement is written during the beginning of the program and whenever now FACTORIAL appears in an instruction or another directive, the assembler substitutes the value 5.
- The advantage of using EQU in this manner is that if FACTORIAL is used several times in a program and the value has to be changed, all that has to change the EQU statement and reassemble the program.
- The assembler with automatically put the new value each time it finds the name FACTORIAL.

6. Define Byte [DB]

- This directive defines the byte type variable.

- It is also useful to set one or more storage locations aside.
- The format of this directive is as follows :

  [name] DB initial value
- The initial value can be a numerical value (8 – bit long) or more than one 8 bit numeric values.
- It can be a constant expression, or a string constant or even a question mark.
- The initial value can be a signed or unsigned number. Its range is from – 128 to + 127 if unsigned and 0 to 255 if it is unsigned.

7. **Define Word or Word [DW]**
   - The DW directive defines items that are one word (two bytes) in length.
   - For unsigned numeric data the range of values is 0 to 65, 535
   - For signed data the range of values is – 32, 768 to + 32, 767.
   - Its format is

     [name] DW initial value.

     e.g. List DW 2534.

8. **Define Double Word or DWORD [DD]**
   - It defines the data items that are a double word (four bytes) in length.
   - It creates storage for 32 bit double words. The format is

     [name] DD initial value.

     e.g. BUFF DD ?

9. **Define Quad Word or QWORD [DQ]**
   - This directive is used to tell the assembler to declare variable 4 words in length or to reserve 4 words of storage in memory.
   - It may define one or more constants, each with a maximum of 8 bytes or 16 Hex digits.
   - Its format is :

     [name] DQ initial value, [initial value].

     e.g. Num DQ 1234567898765432H.

10. **Define Ten Bytes or TBYTE [DT]**
    - It is used to define the data items that are 10 bytes long.
    - Its format is

      [name] DT initial value, [initial value].

      e.g. unpack DT 1234567890.
    - Unlike the other data directives with store hexadecimal numbers, DT will directly store the data in decimal form.

11. **ORG-Originate**
    - The ORG directive allows us to set the location counter to any desired value at any point in the program.
    - The location counter is automatically set to 0000H when the assembler reads a segment.
    - Its format is ORG expression.

      e.g. ORG 500 H ; Set the location counter to 500 H.

A $ is used to represent the current value of LC.

- The $ represents the next available byte location where assembler can put a data or code byte.
- The $ is often used in ORG statements to inform the assembler to make change in location counter relative to its current value.
- e.g. ORG $ + 50 ; Increments the location counter by 50 from its ; current value.

ASSUME

- The directive is used for telling the assembler the name of the logical segment which should be used.
- The format of the assume directive is as follows,

  ASSUME segment register : segment-name :
- The segment register can be CS, DS, SS and ES.
- The example of Assume directive is as follows,

  ASSUME CS Code, DS : Data, SS : Stack :
- ASSUME statement can assign upto 4 segment registers in any sequence.
- In this example, DS : Data means that the assembler is to associate the name of data segment with DS register.
- Similarly CS : Code tells the assembler to associate the name of code segment with CS register and so on.

13. END

- This is placed at the end of a source and it acts as the last statement of a program.
- This is because the END directive terminates the entire program.
- The assembler will neglect any statement after an END directive.
- The format of END directive is as follows :

  END

14. SEGMENT and ENDS

- The SEGMENT directive is used to indicate the start of a logical statement.
- ENDS directive is used with the segment directive.
- ENDS directive indicates the end of the segment.
- Its format is

```
name SEGMENT        ; Begin Segment
name ENDS           ; End Segment.
e.g. : DATA SEGMENT
DATA ENDS.
```

15. GROUP

- This directive collects the segments of the same type under one name.
- It does it so that the segments that are grouped will reside within one segment usually data segment.
- Its format is

  [name] GROUP Seg-name, [seg-name]

  e.g. : NAME GROUP SEG 1, SEG 2.

74

16. MACRO AND ENDM

- The macros in the programs can be defined by MACRO directive.
- The ENDM directive is used along with the Macro directive.
- ENDM defines the end of macro.
- Its format is

```
DISP MACRO
                ; Statements insιde the Macro
ENDM
```

17. ALIGN

- This directive will tell the assembler to align the next instruction on an address which corresponds to the given value.
- Such an alignment will allow the proçessor to access words and double words.

$\text{ALIGN} \quad \underbrace{\text{number}} \rightarrow$ This number should be 2, 4, 8, 16 .... i.e it should be a power of 2.

- The example of Align directive are ALIGN 2 and ALIGN 4.
- ALIGN 2 is used for starting the data segment on a word boundary whereas ALIGN 4 will start the data segment on a double boundary word.

18. EVEN (Align on Even Memory Address)

- It tells the assembler to increment its location counter if required, so that the next defined data item is aligned on an even storage boundary.
- The 8086 can read a word from memory in one bus cycle if the word is at an even address.
- If the word starts at an odd address, the microprocessor must do two read cycles to get 2 bytes of the word. In the first cycle it will read the LSB and in the second it will read MSB
- Therefore, a series of even words can be read more quickly if they are at an even address.

```
e.g. : EVEN   TABLE   DB   10 DUP (O)  ;  It declares an array named
                                       ;  TABLE of 10 bytes which are
                                       ;  starting from an evenaddress.
```

19. LABEL

- This directive assigns name to the current value of the Location Counter.
- The LABEL directive must be followed by a term which specifies the type associated with that symbolic name.
- If label is going to be used as the destination for a jump or call, then the LABEL must be specified as type near or type Far.
- If the label is going to be used to reference a data item, then the label must be specified as type byte, word or double word.

```
e.g. : STACK SEGMENT
DW 50 DUP (O)              ;  Set aside 50 words for stack.
STACK_TOP    LABEL WORD ;  Give a symbolic name to next location
                           ;  after the last word in stack STACK ENDS
```

20. PROC-Procedure

- This directive is used to indicate the start of a procedure.
- The procedures are of two types NEAR and FAR.
- If the procedure is within the same segment then the label NEAR should be given after procedure.
- If the procedure is in another module then the label FAR should be given after procedure.
- Its format is **[procedure-name]**
  PROC NEAR.

21. EXTRN

- It indicates that the names or labels that follow the EXTRN directive are in some other assembly module.
  e.g. : EXTRN DISP : FAR.
  The statement tells the assembler that DISP is a label of type far in another assembly module.
- To call a procedure that is in a program module assembled at a different time from that which contains the CALL instruction, the assembler has to be told that the procedure is external.
- The assembler will then put information in the object code file so that linker can connect the two modules together.
- The names or labels that are external in one module must be declared public with the PUBLIC directive in the module where they are defined.
- Its format is

```
e.g. : Procedure_HereSegment
       EXTRN FACT     :  FAR
       XTRN SUM       :  NEAR.
     Procedure_HereEnds.
```

22. PUBLIC

- It informs the assembler and linker that the identified variables in a program are to be referenced by other modules linked with the current one.
- Its format is
- PUBLIC variable, [variable]
- The variable can be a number (up to two bytes) or a label or a symbol.
  e.g. PUBLIC MULTIPLIER, DIVISOR.

This makes the two variables Multiplier and Divisor available to other assembly modules.

23. PAGE

- This directive is used to specify the maximum number of lines on a page and the maximum number of characters on a line.
- The format of this directive is as follows :

PAGE [length], [width]

```
        |            └──→ Number of characters on a line.
        └──→ Number of lines on a page
```

- The example is PAGE 55, 102 which shows that the, there are 55 lines per page and 102 characters per line.
- The number of lines per page typically range from 10 to 255 and the number of characters per line will range from 60 to 132.

24. TITLE

- It helps the user for controlling the format of listing of an assembled program.
- It is used to give a title to program and print the title on the second line of each page of the program.
- The maximum number of characters allowed as a title is 60.
- The format of this assembler directive is as follows,

  TITLE Text

25. INCLUDE

- This directive is used to tell the assembler to insert a block of source code from the named file into the current source module. This shortens the source code.
- Its format is

  INCLUDE path : file name.

26. NAME

This directive assigns a specific name to each assembly module when programs consisting of several modules are written.

27. DUP Operator

- Whenever we want to allocate space for a table or an array the DUP directive can be used. The DUP operator it will be used after a storage allocation directive like (DB, DW, DQ, DT, DD).
- With DUP, we can repeat one or more values while assigning the storage values .
- Its format is

  [name] Data-Type Number DUP (value).

e.g. : List DB 20 DUP [0] ; A list of 20 bytes, where each byte is zero.

A DUP operator may be nested.

e.g. LIST 1 DB 4 DUP (4 DUP [0]. 3 DUP ['X']) ; The assembler assigns the values in memory here as follows,

00, 00, 00, 00, x, x, x
00, 00, 00, 00, x, x, x
00, 00, 00, 00, x, x, x
00, 00, 00, 00, x, x, x

28. GLOBAL – Declare Symbols as PUBLIC or EXTRN

- This directive can be used instead of PUBLIC directive or instead EXTRN directive.
- For a name or variable defined in the current module, the GLOBAL directive is used to make the variable available to all other modules.
- Its format is

e.g. : GLOBAL FACTOR ; it makes the variable FACTOR public so that it can
; be accessed from all other modules that are linked .

GLOBAL FACTOR : WORD ; it tells assembler that variable FACTOR of
; type word which is in another assembly module
; or EXTRN.

29. LENGTH

- It is an operator.
- It informs the assembler to find the number of elements in a named data item like a string or an array.
- The length of string is always stored in Hex by the 8086 .
- Its format is :

e.g. : MOV CX, LENGTH STRING ; Loads the Length of string in CX.

30. OFFSET

- It is an operator.
- It informs the assembler to determine the offset or displacement of a named data item.
- It may also determine the offset of a procedure from the start of the segment which contains it
- Its format is :

e.g. : MOV AX, OFFSET NUM ; It will load the offset of NUM in the AX
; register.

31. ENDP-End Procedure

- This directive is used along with the name of the procedure to indicate the end of procedure
- ENDP defines the end of procedure.

```
e.g. : FACT PROC FAR    ; Start procedure named FACT and informs the
                        ; assembler that the procedure is FAR.
       Body of Procedure
       FACT ENDP        ; End of Procedure FACT.
```

32. PTR-Pointer

- The pointer is an operator.
- It is used to assign a specific type to a variable or to a label.
- It is necessary to do this in any instruction where the type of operand is not clear.

e.g. (i) INC [AX] ; The assembler does not know whether to increment the byte
; pointed by BX or increment the word pointed to by BX.

This difficulty could be avoided by using PTR directive.

e.g. INC BYTE PTR [BX] : it increments byte pointed to by [BX]
INC WORD PTR [BX] : it increments word pointed to by [BX].

(ii) The PTR operator can be used to override the declared type of variable.

e.g. WORD DB 23, 45, 10, 56.

The access to the array WORDS will be in terms of bytes.

MOV AL, BYTE PTR WORDS.

(iii) During indirect jump instruction, PTR could be used JMP [BX]. This instruction does not tell the assembler whether to code the instruction for a