UNIT 2 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Structure **Page No.**

2.0 INTRODUCTION

In the previous unit, we have discussed the 8086 microprocessor. We have discussed the register set, instruction set and addressing modes for this microprocessor. In this and two later units we will discuss the assembly language for 8086/8088 microprocessor. Unit 1 is the basic building block, which will help in better understanding of the assembly language. In this unit, we will discuss the importance of assembly language, basic components of an assembly program followed by discussions on the program developmental tools available. We will then discuss what are COM programs and EXE programs. Finally we will present a complete example. For all our discussions, we have used Microsoft Assembler (MASM). However, for different assemblers the assembly language directives may change. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using.

2.1 OBJECTIVES

After going through this unit you should be able to:

- define the need and importance of an assembly program;
- define the various directives used in assembly program;
- write a very simple assembly program with simple input – output services;
- define COM and EXE programs; and
- differentiate between COM and EXE programs.

2.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

Machine language code consists of the 0-1 combinations that the computer decodes directly. However, the machine language has the following problems:

- It greatly depends on machine and is difficult for most people to write in 0-1 forms.
- DEBUGGING is difficult.
- Deciphering the machine code is very difficult. Thus program logic will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instruction of a machine. This symbolic code for each instruction is called a mnemonic. The mnemonic for a particular instruction consists of letters that suggest the operation to be performed by that instruction. For example, ADD mnemonic is used for adding two numbers. Using these mnemonics machine language instructions can be written in symbolic form with each machine instruction represented by one equivalent symbolic instruction. This is called an assembly language.

Pros and Cons of Assembly Language

The following are some of the advantages / disadvantages of using assembly language:

- Assembly Language provides more control over handling particular hardware and software, as it allows you to study the instructions set, addressing modes, interrupts etc.
- Assembly Programming generates smaller, more compact executable modules: as the programs are closer to machine, you may be able to write highly optimised programs. This results in faster execution of programs.

Assembly language programs are at least 30% denser than the same programs written in high-level language. The reason for this is that as of today the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line of code for a single instruction. This will be true especially in case of string related programs.

On the other hand assembly language is machine dependent. Each microprocessor has its own set of instructions. Thus, assembly programs are not portable.

Assembly language has very few restrictions or rules; nearly everything is left to the discretion of the programmer. This gives lots of freedom to programmers in construction of their system.

Uses of Assembly Language

Assembly language is used primarily for writing short, specific, efficient interfacing modules/ subroutines. The basic idea of using assembly is to support the HLL with some highly efficient but non–portable routines. It will be worth mentioning here that UNIX mostly is written in C but has about 5-10% machine dependent assembly code. Similarly in telecommunication application assembly routine exists for enhancing efficiency.

2.3 ASSEMBLY PROGRAM EXECUTION

An assembly program is written according to a strict set of rules. An editor or word processor is used for keying an assembly program into the computer as a file, and then the assembler is used to translate the program into machine code.

There are 2 ways of converting an assembly language program into machine language:

1) Manual assembly

2) By using an assembler.

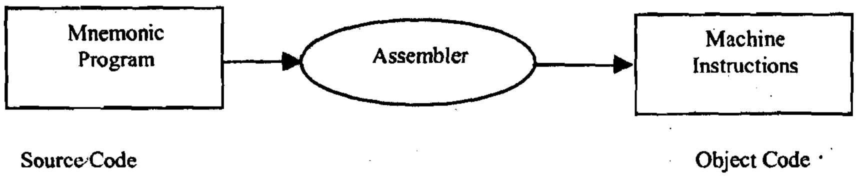
Manual Assembly

It was an old method that required the programmer to translate each opcode into its numerical machine language representation by looking up a table of the microprocessor instructions set, which contains both assembly and machine language instructions. Manual assembly is acceptable for short programs but becomes very inconvenient for large programs. The Intel SDK-85 and most of the earlier university kits were programmed using manual assembly.

Using an Assembler

The symbolic instructions that you code in assembly language is known as - Source program.

An assembler program translates the source program into machine code, which is known as object program.



The steps required to assemble, link and execute a program are:

Step 1: The assembly step involves translating the source code into object code and generating an intermediate .OBJ (object file) or module.

The assembler also creates a header immediately in front of the generated .OBJ module; part of the header contains information about incomplete addresses. The .OBJ module is not quite in executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linker's tasks include completing any address left open by the assembler and combining separately assembled programs into one executable module.

The linker:

- combines assembled module into one executable program
- generates an .EXE module and initializes with special instructions to facilitate its subsequent loading for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to load in memory, it is now able to resolve any remaining address still left incomplete in the header. The loader drops the header and creates a program segment prefix (PSP) immediately before the program is loaded in memory.

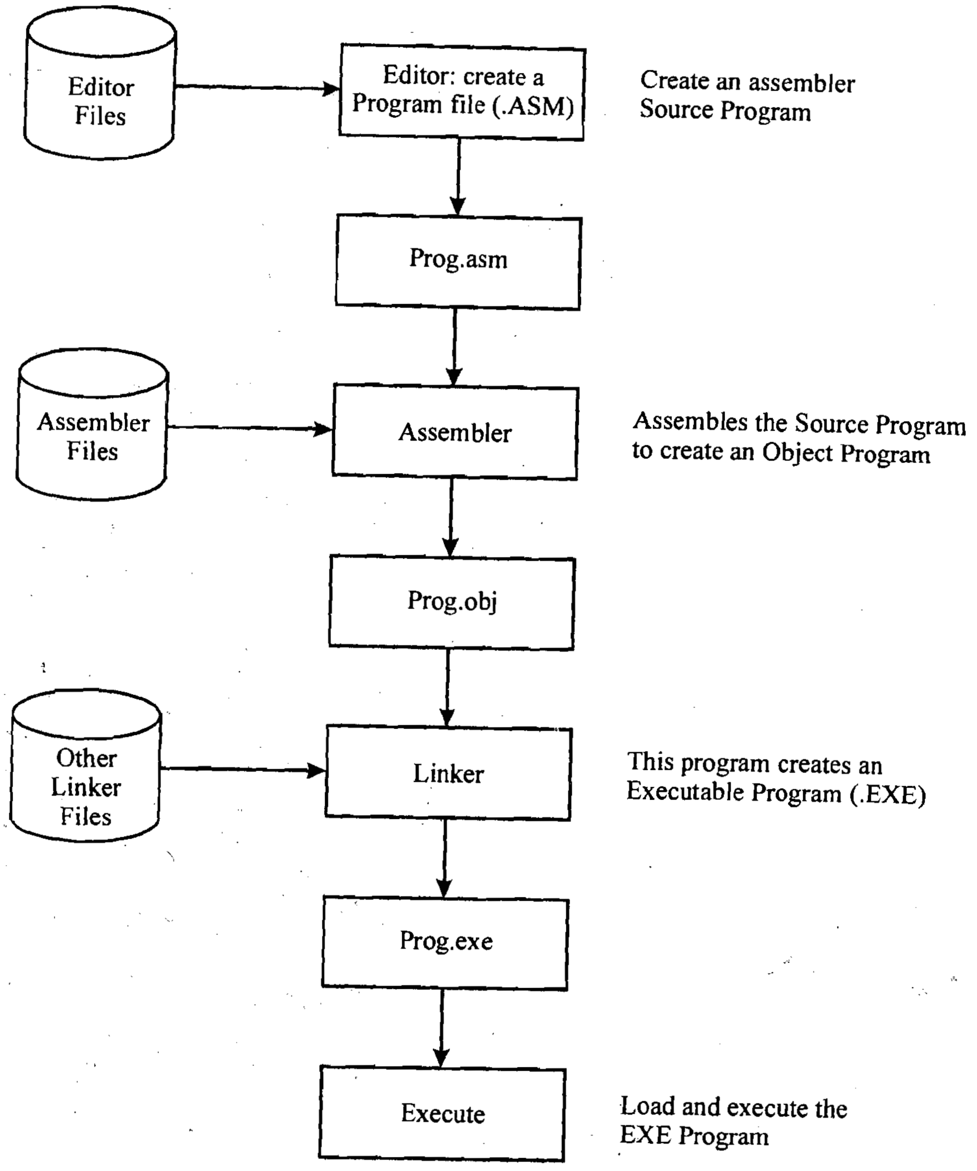

Editor Files
Editor: create a Program file (.ASM)
Create an assembler Source Program
Prog.asm
Assembler Files
Assembler
Assembles the Source Program to create an Object Program
Prog.obj
Other Linker Files
Linker
This program creates an Executable Program (.EXE)
Prog.exe
Execute
Load and execute the EXE Program

Figure 2: Program Assembly

All this conversion and execution of Assembly language performed by Two-pass assembler.

Two-pass assembler: Assemblers typically make two or more passes through a source program in order to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction.

Pass 1: Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, name of data fields and programs labels and their relative location (offset) within the segment.

Pass 1 determines the amount of code to be generated for each instruction.

Pass 2: The assembler uses the symbol table that it constructed in Pass 1. Now it knows the length and relative position of each data field and instruction, it can complete the object code for each instruction. It produces .OBJ (Object file), .LST (list file) and cross reference (.CRF) files.

Tools required for assembly language programming

The tools of the assembly process described below may vary in details.

Editor

The editor is a program that allows the user to enter, modify, and store a group of instructions or text under a file name. The editor programs can be classified in 2 groups.

- Line editors
- Full screen editors.

Line editors, such as EDIT in MS DOS, work with the manage one line at a time. Full screen editors, such as Notepad, Wordpad etc. manage the full screen or a paragraph at a time. To write text, the user must call the editor under the control of the operating system. As soon as the editor program is transferred from the disk to the system memory, the program control is transferred from the operating system to the editor program. The editor has its own command and the user can enter and modify text by using those commands. Some editor programs such as WordPerfect are very easy to use. At the completion of writing a program, the exit command of the editor program will save the program on the disk under the file name and will transfer the control to the operating system. If the source file is intended to be a program in the 8086 assembly language the user should follow the syntax of the assembly language and the rules of the assembler.

Assembler

An assembly program is used to transfer assembly language mnemonics to the binary code for each instruction, after the complete program has been written, with the help of an editor it is then assembled with the help of an assembler.

An assembler works in 2 phases, i.e., it reads your source code two times. In the first pass the assembler collects all the symbols defined in the program, along with their offsets in symbol table. On the second pass through the source program, it produces binary code for each instruction of the program, and give all the symbols an offset with respect to the segment from the symbol table.

The assembler generates three files. The object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled with no errors. The errors that are detected by the assembler are called the symbol errors. For example,

```
MOVE AX1, ZX1 ;
```

In the statement, it reads the word MOVE, it tries to match with the mnemonic sets, as there is no mnemonic with this spelling, it assumes it to be an identifier and looks for its entry in the symbol table. It does not even find it there therefore gives an error as undeclared identifier.

List file is optional and contains the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for purely documentation purposes. Some of the assemblers available on PC are MASM, TURBO etc.

Linker

For modularity of your programs, it is better to break your program into several sub routines. It is even better to put the common routine, like reading a hexadecimal number, writing hexadecimal number, etc., which could be used by a lot of your other programs into a separate file. These files are assembled separately. After each file has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines can be linked to your other program also. The program that links your program is called the linker.

The linker produces a link file, which contains the binary code for all compound modules. The linker also produces link maps, which contains the address information about the linked files. The linker however does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. This form a program is said to be relocatable because it can be put anywhere in memory to be run.

Loader

Loader is a program which assigns absolute addresses to the program. These addresses are generated by adding the address from where the program is loaded into the memory to all the offsets. Loader comes into action when you want to execute your program. This program is brought from the secondary memory like disk. The file name extension for loading is .exe or .com, which after loading can be executed by the CPU.

Debugger

The debugger is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions.

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program and display register contents after the execution.
- Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., convert the object code into the source code or mnemonics.

In summary, to run an assembly program you may require your computer:

- A word processor like notepad
- MASM, TASM or Emulator
- LINK.EXE, it may be included in the assembler
- DEBUG.COM for debugging if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

a. Programming errors: They are the familiar errors you can encounter in the course of executing a program written in any language.
b. System errors: These are unique to assembly language that permit low-level operations. A system error is one that corrupts or destroys the system under which the program is running - In assembly language there is no supervising

interpreter or compiler to prevent a program from erasing itself or even from erasing the computer operating system.

2.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

Sample Program

In this program we just display:

```
Line    Offset
Numbers        ————————  Source Code ————————
0001           DATA SEGMENT
0002    0000         MESSAGE DB "HAVE A NICE DAY!$"
0003           DATA ENDS
0004           STACK SEGMENT
0005                 STACK 0400H
0006           STACK ENDS
0007           CODE SEGMENT
0008                 ASSUME   CS: CODE, DS: DATA   SS: STACK
0009    Offset Machine  Code
0010    0000   B8XXXX       MOV AX, DATA
0011    0003   8ED8         MOV DS, AX
0012    0005   BAXXXX       MOV DX, OFFSET MESSAGE
0013    0008   B409         MOV AH, 09H
0014    000A   CD21         INT  21H
0015    000C   B8004C       MOV AX,  4C00H
0016    000F   CD21         INT  21H
0017                        CODE ENDS
0018                        END
```

The details of this program are:

2.4.1 The Program Annotation

The program annotation consists of 3 columns of data: line numbers, offset and machine code.

- The assembler assigns line numbers to the statements in the source file sequentially. If the assembler issues an error message; the message will contain a reference to one of these line numbers.

- The second column from the left contains offsets. Each offset indicates the address of an instruction or a datum as an offset from the base of its logical segment, e.g., the statement at line 0010 produces machine language at offset 0000H of the CODE SEGMENT and the statement at line number 0002 produces machine language at offset 0000H of the DATA SEGMENT.

- The third column in the annotation displays the machine language produce by code instruction in the program.

Segment numbers: There is a good reason for not leaving the determination of segment numbers up to the assembler. It allows programs written in 8086 assembly language to be almost entirely relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Have a nice day$" somewhere in memory. It is located in the DATA SEGMENT. Since the characters are stored in ASCII, therefore it will occupy 15 bytes (please note each blank is also a character) in the DATA SEGMENT.

Missing offset: The xxxx in the machine language for the instruction at line 0010 is there because the assembler does not know the DATA segment location that will be determined at loading time. The loader must supply that value.

Program Source Code

Each assembly language statement appears as:

{identifier} Keyword {{parameter},} {;comment}.

The element of a statement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a carriage return, a line feed.

Keyword: A keyword is a statement that defines the nature of that statement. If the statement is a directive then the keyword will be the title of that directive; if the statement is a data-allocation statement the keyword will be a data definition type. Some examples of the keywords are: SEGMENT (directive), MOV (statement) etc.

Identifiers: An identifier is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are name and label.

1. Name refers to the address of a data item such as counter, arr etc.
2. Label refers to the address of our instruction, process or segment. For example MAIN is the label for a process as:

```
MAIN PROC FAR
A20:  BL,45 ; defines a label A20.
```

Identifier can use alphabet, digit or special character but it always starts with an alphabet.

Parameters: A parameter extends and refines the meaning that the assembler attributes to the keyword in a statement. The number of parameters is dependent on the Statement.

Comments: A comment is a string of a text that serves only as internal document action for a program. A semicolon identifies all subsequent text in a statement as a comment.

2.4.2 Directives

Assembly languages support a number of statements. This enables you to control the way in which a source program assembles and list. These statements, called directives, act only when the assembly is in progress and generate no machine-executable code. Let us discuss some common directives.

1. **List:** A list directive causes the assembler to produce an annotated listing on the printer, the video screen, a disk drive or some combination of the three. An annotated listing shows the text of the assembly language programs, numbers of each statement in the program and the offset associated with each instruction and each datum. The advantage of list directive is that it produces much more informative output.

2. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. That statement directs the assembler to treat tokens in the

source file that begins with a dollar sign as numeric constants in hexadecimal notation.

3. **PROC Directive:** The code segment contains the executable code for a program, which consists of one or more procedures defined initially with the PROC directive and ended with the ENDP directive.

```
Procedure-name   PROC   FAR   ; Beginning of Procedure
Procedure-name   ENDP   FAR   ; End Procedure
```

4. **END DIRECTIVE:** ENDS directive ends a segment, ENDP directive ends a procedure and END directive ends the entire program that appears as the last statement.

5. **ASSUME Directive:** An .EXE program uses the SS register to address the base of stack, DS to address the base of data segment, CS to address base of the code segment and ES register to address the base of Extra segment. This directive tells the assembler to correlate segment register with a segment name. For example,

```
ASSUME  SS: stack_seg_name, DS: data_seg_name,  CS: code_seg_name.
```

6. **SEGMENT Directive:** The segment directive defines the logical segment to which subsequent instructions or data allocations statement belong. It also gives a segment name to the base of that segment.

The address of every element in a 8086 assembly program must be represented in segment - relative format. That means that every address must be expressed in terms of a segment register and an offset from the base of the segmented addressed by that register. By defining the base of a logical segment, a segment directive makes it possible to set a segment register to address that base and also makes it possible to calculate the offset of each element in that segment from a common base.

An 8086 assembly language program consists of logical segments that can be a code segment, a stack segment, a data segment, and an extra segment.

A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

CODE SEGMENT

The logical program segment is named code segment. When the linker links a program it makes a note in the header section of the program's executable file describing the location of the code segment when the DOS invokes the loader to load an executable file into memory, the loader reads that note. As it loads the program into memory, the loader also makes notes to itself of exactly where in memory it actually places each of the program's other logical segments. As the loader hands execution over to the program it has just loaded, it sets the CS register to address the base of the segment identified by the linker as the code segment. This renders every instruction in the code segment addressable in segment relative terms in the form CS: xxxx.

The linker also assumes by default that the first instruction in the code segment is intended to be the first instruction to be executed. That instruction will appear in memory at an offset of 0000H from the base of the code segment, so the linker passes that value on to the loader by leaving an another note in the header of the program's executable file.

The loader sets the IP (Instruction Pointer) register to that value. This sets CS:IP to the segment relative address of the first instruction in the program.

STACK SEGMENT

8086 Microprocessor supports the **Word stack**. The stack segment parameters tell the assembler to alert the linker that this segment statement defines the program stack area.

A program must have a stack area in that the computer is continuously carrying on several background operations that are completely transparent, even to an assembly language programmer, for example, a real time clock. Every 55 milliseconds the real time clock interrupts. Every 55 ms the CPU is interrupted. The CPU records the state of its registers and then goes about updating the system clock. When it finishes servicing the system clock, it has to restore the registers and go back to doing whatever it was doing when the interruption occurred. All such information gets recorded in the stack. If your program has no stack and if the real time clock were to pulse while the CPU is running your program, there would be no way for the CPU to find the way back to your program when it was through updating the clock. 0400H byte is the default size of allocation of stack. Please note if you have not specified the stack segment it is automatically created.

DATA SEGMENT

It contains the data allocation statements for a program. This segment is very useful as it shows the data organization.

Defining Types of Data

The following format is used for defining data definition:

Format for data definition:

{Name} <Directive> <expression>
Name - a program references the data item through the name although it is optional.
Directive: Specifying the data type of assembly.
Expression: Represent a value or evaluated to value.

The list of directives are given below:

Directive	Description	Number of Bytes
DB	Define byte	1
DW	Define word	2
DD	Define double word	4
DQ	Define Quad word	8
DT	Define 10 bytes	10

DUP Directive is used to duplicate the basic data definition to 'n' number of times

```
   ARRAY            DB          10 DUP (0)
```

In the above statement ARRAY is the name of the data item, which is of byte type (DB). This array contains 10 duplicate zero values; that is 10 zero values.

EQU directive is used to define a name to a constant

```
        CONST       EQU         20
```

Type of number used in data statements can be octal, binary, haxadecimal, decimal and ASCII. The above statement defines a name CONST to a value 20.

Some other examples of using these directives are:

```
TEMP      DB   0111001B     ; Binary value in byte operand
                                    ; named temp
VALI      DW   7341Q        ; Octal value assigned to word
                                    ; variable
Decimal   DB   49           ; Decimal value 49 contained in
                                    ; byte variable
HEX       DW   03B2AH       ; Hex decimal value in word
                                    ; operand
ASCII     DB   'EXAMPLE'    ; ASCII array of values.
```

Check Your Progress 1

1. Why should we learn assembly language?

   ....

   ....

   ....

2. What is a segment? Write all four main segment names.

   ....

   ....

   ....

3. State True or False. T F

   (a) The directive DT defines a quadword in the memory ☐

   (b) DUP directive is used to indicate if a same memory location is used by two different variables name. ☐

   (c) EQU directive assign a name to a constant value. ☐

   (d) The maximum number of active segments at a time in 8086 can be four. ☐

   (e) ASSUME directive specifies the physical address for the data values of instruction. ☐

   (f) A statement after the END directive is ignored by the assembler. ☐

2.5 INPUT OUTPUT IN ASSEMBLY PROGRAM

A software interrupt is a call to an Interrupt servicing program located in the operating system. Usually the input-output routine in 8086 is constructed using these interrupts.

2.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are: keyboard, printer, monitor, an error condition, trap etc.

8086 recognizes two kinds of interrupts: **Hardware** interrupts and **Software** interrupts.

Hardware interrupts are generated when a peripheral Interrupt servicing program requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

Let us discuss the software interrupts in more detail. A software interrupt is initiated using the following statements:

INT number

In 8086, this interrupt instruction is processing using the **interrupt vector table (IVT).** The IVT is located in the first 1K bytes of memory, and has a total of 256 entities, each of 4 bytes. An entry in the interrupt vector table is identified by the number given in the interrupt instruction. The entry stores the address of the operating system subroutine that is used to process the interrupt. This address may be different for different machines. Figure 1 shows the processing of an interrupt.

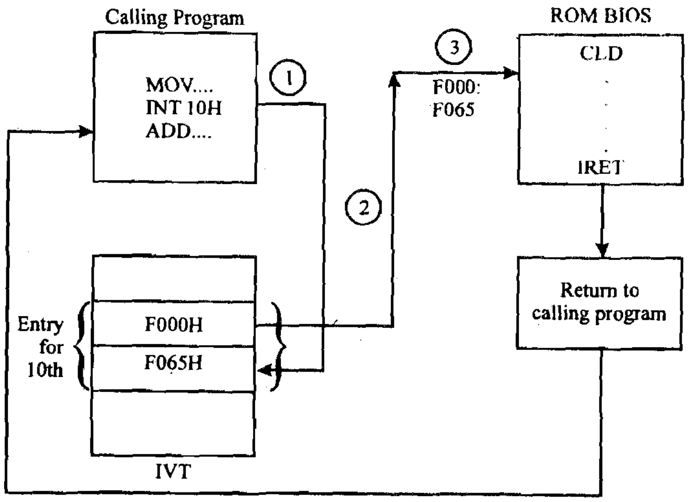


Figure 1: Processing of an Interrupt

The interrupt is processed as:

Step 1: The number field in INT instruction is multiplied by 4 to find its entry in the interrupt vector table. For example, the IVT entry for instruction INT 10h will be found at IVT at an address 40h. Similarly the entry of INT 3h will be placed at 0Ch.

Step 2: The CPU locates the interrupt servicing routine (ISR) whose address is stored at IVT entry of the interrupt. For example, in the figure above the ISR of INT 10h is stored at location at a segment address F000h and an offset F065h.

Step 3: The CPU loads the CS register and the IP register, with this new address in the IVT, and transfers the control to that address, just like a far CALL, (discussed in the unit 4).

Step 4: IRET (interrupt return) causes the program to resume execution at the next instruction in the calling program.

Keyboard Input and Video output

A Keystroke read from the keyboard is called a console input and a character displayed on the video screen is called a console output. In assembly language, reading and displaying character is most tedious to program. However, these tasks were greatly simplified by the convenient architecture of the 8086/8088. That architecture provides for a pack of software interrupt vectors beginning at address 0000:0000.

The advantage of this type of call is that it appears static to a programmer but flexible to a system design engineer. For example, INT 00H is a special system level vector that points to the "recovery from division by zero" subroutine. If new designer come and want to move interrupt location in memory, it adjusts the entry in the IVT vector of interrupt 00H to a new location. Thus from the system programmer point of view, it is relatively easy to change the vectors under program control.

One of the most commonly used Interrupts for Input /Output is called DOS function call. Let us discuss more about it in the next subsection:

2.5.2 DOS Function Calls (Using INT 21H)

INT 21H supports about 100 different functions. A function is identified by putting the function number in the AH register. For example, if we want to call function number 01, then we place this value in AH register first by using MOV instruction and then call INT 21H:

Some important DOS function calls are:

DOS Function Call	Purpose	Example
AH = 01H	For reading a single character from keyboard and echo it on monitor. The input value is put in AL register.	To get one character input in a variable in data segment you may include the following in the code segment: MOV AH,01 INT 21H MOV X, AL (Please note that interrupt call will return value in AL which is being transferred to variable of data segment X. X must be byte type).
AH = 02H	This function prints 8 bit data (normally ASCII) that is stored in DL register on the screen.	To print a character let say '?' on the screen we may have to use following set of commands: MOV AH, 02H; MOV DL, '?' INT 21H
AH = 08H	This is an input function for inputting one character. This is same as AH = 01H functions with the only difference that value does not get displayed on the screen.	Same example as 01 can be used only difference in this case would be that the input character wouldn't get displayed MOV AH, 08H INT 21H MOV X, AL
AH = 09H	This program outputs a string whose offset is stored in DX register and that is terminated using a $ character. One can print newline, tab character also.	To print a string "hello world" followed by a carriage return (control character) we may have to use the following assembly program segment.

Example of AH = 09H	CR EQU ODH ; ASCII code of carriage return. DATA SEGMENT STRING DB 'HELLO WORLD', CR, '$' DATA ENDS CODE SEGMENT : MOV AX, DATA MOV DS, AX MOV AH, 09H MOV DX, OFFSET STRING ; Store the offset of string in DX register. INT 21H	
AH = 0AH	For input of string up to 255 characters. The string is stored in a buffer.	Look in the examples given.
AH = 4CH	Return to DOS	

Some examples of Input

(i) Input a single ASCII character into BL register without echo on screen

```
CODE SEGMENT

  MOV   AH, 08H  ; Function 08H
  INT   21H      ; The character input in AL is
  MOV   BL, AL   ; transfer to BL
  :
CODE ENDS
```

(ii) Input a Single Digit for example (0,1, 2, 3, 4, 5, 6, 7, 8, 9)

```
CODE SEGMENT
      ...
; Read a single digit in BL register with echo. No error check in the Program
        MOV   AH,  01H

        INT   21H
; Assuming that the value entered is digit, then its ASCII will be stored in AL.
; Suppose the key pressed is 1 then ASCII '31' is stored in the AL. To get the
; digit 1 in AL subtract the ASCII value '0' from the AL register.
; Here it store      0 as ASCII 30,
; 1 as 31,  2 as 32.......9 as 39
; to store 1 in memory subtract 30 to get 31 – 30  = 1
        MOV BL, AL
        SUB  BL, ' 0'  ;  ' 0'  is digit 0 ASCII
             ; OR
        SUB   BL,  30H
; Now BL contain the single digit 0 to 9
; The only code missing here is to check whether the input is in the specific
; range.
...
CODE ENDS.
```

(iii) Input numbers like (10, 11.............99)

```
; If we want to store 39, it is actually 30 + 9
; and it is 3 × 10 + 9
; to input this value through keyboard, first we input the tenth digit e.g., 3 and
; then type 9
                MOV   AH, 08H
                INT   21H
                MOV  BL, AL ; If we have input 39 then, BL will first have character
; 3, we can convert it to 3 using previous logic that is 33 – 30 = 3.
                SUB   BL, '0'
                MUL   BL, AH          ; To get 30 Multiply it by 10.
                                      ; Now BL Store 30
                                      ; Input another digit from keyboard
                MOV   AH,  08H
                INT   21H;
                MOV   DL, AL          ; Store AL in DL
                SUB   DL, '0'         ;    (39 – 30) = 9.
; Now BL contains the value: 30 and DL has the value 9 add them and get the
; required numbers.
                ADD   BL, DL
; Now BL store 39.  We have 2 digit value in BL.
```

Let us try to summarize these segments as:

```
CODE SEGMENT
;  Set DS register
  MOV AX, DATA    ;⎫ boiler plate code to set the DS register so that the
  MOV  DS, AX     ;⎭ program can access the data segment.

; read first digit from keyboard
                MOV     AH,  08
                INT     21H
                MOV     BL, AL
                SUB     BL, '0'
                MUL     BL,  10H
; read second digit from keyboard
                MOV     AH, 08H
                INT     21H
                MOV     DL, AL
                SUB     DL, '0'
; DL = 9 AND BL = 30
                SUM      BL, DL
; now BL store 39
CODE ENDS.
```

Note: Boilerplate code is the code that is present more or less in the same form in every assembly language program.

Strings Input

```
CODE SEGMENT
        . . .
        MOV AH, 0AH  ; Move 04 to AH register
        MOV DX, BUFF         ; BUFF must be defined in data segment.
        INT   21H
.....
CODE ENDS
DATA  SEGMENT
        BUFF  DB       50       ; max length of string,
                                ; including CR, 50 characters
              DB       ?        ; actual length of string not known at present
              DB 50 DUP(0)      ; buffer having 0 values
DATA ENDS.
```

Explanation

The above DATA segment creates an input buffer BUFF of maximum 50 characters. On input of data 'JAIN' followed by enter data would be stored as:

50	4	J	A	I	N	#

Examples of Display on Video Monitor

(1) **Displaying a single character**

```
; display contents of BL register (assume that it has a single character)
MOV  AH, 02H
MOV  DL, BL.
INT   21 H.
```

Here data from BL is moved to DL and then data display on monitor function is called which displays the contents of DL register.

(2) **Displaying a single digit (0 to 9)**

Assume that a value 5 is stored in BL register, then to output BL as ASCII value add character '0' to it

```
ADD  BL, '0'
MOV  AH, 02H
MOV  DL, BL
INT   21H
```

(3) **Displaying a number (10 to 99)**

Assuming that the two digit number 59 is stored as number 5 in BH and number 9 in BL, to convert them to equivalent ASCII we will add '0' to each of them.

```
ADD  BH, '0'
ADD  BL, '0'
MOV  AH, 02H
MOV  DL, BH
INT   21H
MOV  DL, BL
INT   21H
```

(4) **Displaying a string**

```
MOV  AH, 09H
MOV  DX, OFFSET  BUFF
INT   21H
```

Here data in input buffer stored in data segment is going to be displayed on the monitor.

A complete program:

Input a letter from keyboard and respond. "The letter you typed is ___".

```
CODE SEGMENT
;       set the DS register
            MOV  AX, DATA
            MOV DS,  AX
;       Read Keyboard
            MOV  AH,  08H
            INT  21H
;       Save input
            MOV  BL, AL
;       Display first part of Message
            MOV AH, 09H
            MOV DX, OFFSET MESSAGE
            INT  21 H
;       Display character of BL register
            MOV  AH,  02H
            MOV DL, BL
            INT  21 H
;       Exit to DOS
            MOV AX, 4C00H
            INT 21H
CODE ENDS

DATA SEGMENT
        MESSAGE DB "The letter you typed is $"
DATA ENDS
END.
```

2.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

COM Program: Having all the segments as part of one segment
EXE Program: which have more than one segment.

Let us look into brief details of these programs.

2.6.1 COM Programs

A COM (Command) program is the binary image of a machine language program. It is loaded in the memory at the lowest available segment address. The program code begins at an offset 100h, the first 1K locations being occupied by the IVT.

A COM program keeps its code, data, and stack segments within the same segment. Since the offsets in a physical segment can be of 16 bits, therefore the size of COM program is limited to 2^{16} = 64K which includes code, data and stack. The following program shows a COM program:

```
; Title add two numbers and store the result and carry in memory variables.
; name of the segment in this program is chosen to be CSEG

CSEG SEGMENT
      ASSUME CS:CSEG, DS:CSEG, SS:CSEG
      ORG       100h
START:MOV  AX, CSEG         ; Initialise data segment
      MOV DS, AX            ; register using AX
      MOV AL, NUM1          ; Take the first number in AL
```

```
          ADD AL, NUM2              ; Add the 2nd number to it
          MOV RESULT, AL            ; Store the result in location RESULT
          RCL AL, 01                ; Rotate carry into LSB
          AND AL, 00000001B         ; Mask out all but LSB
          MOV CARRY, AL             ; Store the carry result
          MOV AX,4C00h
          INT 21h
          NUM1 DB       15h         ; First number stored here
          NUM2 DB       20h         ; Second number stored here
          RESULT DB     ?           ; Put sum here
          CARRY DB      ?           ; Put any carry here
CSEG ENDS
END START
```

These programs are stored on a disk with an extension .com. A COM program requires less space on disk rather than equivalent EXE program. At run-time the COM program places the stack automatically at the end of the segment, so they use at least one complete segment.

2.6.2 EXE Programs

An EXE program is stored on disk with extension .exe. EXE programs are longer than the COM programs, as each EXE program is associated with an EXE header of 256 bytes followed by a load module containing the program itself. The EXE header contains information for the operating system to calculate the addresses of segments and other components. We will not go into such details in this unit.

The load module of EXE program consists of up to 64K segments, although at the most only four segments may be active at any time. The segments may be of variable size, with maximum size being 64K.

We will write only EXE programs for the following reasons:

- EXE programs are better suited for debugging.
- EXE-format assembler programs are more easily converted into subroutines for high-level languages.
- EXE programs are more easily relocatable. Because, there is no ORG statement, forcing the program to be loaded from a specific address.
- To fully use multitasking operating system, programs must be able to share computer memory and resources. An EXE program is easily able to do this.

An example of equivalent EXE program for the COM program is:

```
; ABSTRACT          this program adds 2 8-bit numbers in the memory locations
;                   NUM1 and NUM2. The result is stored in the
;                   memory location RESULT. If there was a carry
;                   from the addition it will be stored as 0000 0001 in
;                   the location CARRY
; REGISTERS         Uses CS, DS, AX
DATA SEGMENT
          NUM1   DB    15h     ; First number
          NUM2   DB    20h     ; Second number
```

```
    RESULT DB     ?       ; Put sum here
    CARRY  DB     ?       ; Put any carry here
DATA ENDS
CODE SEGMENT
    ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA        ; Initialise data segment
    MOV DS, AX            ; register using AX
    MOV AL, NUM1          ; Bring the first number in AL
    ADD AL, NUM2          ; Add the 2nd number to AL
    MOV RESULT, AL        ; Store the result
    RCL AL, 01            ; Rotate carry into Least Significant Bit (LSB)
    AND AL, 00000001B     ; Mask out all but LSB
    MOV CARRY, AL         ; Store the carry
    MOV AX, 4C00h         ; Terminate to DOS
    INT 21h
CODE ENDS
    END START
```

2.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

Now that we have seen all the details of assembly language programming, let us discuss the art of writing assembly programs in brief.

Preparation of writing the program

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:

```
get NUM1
add NUM2
put sum into memory at RESULT
position carry bit in LSB of byte
            mask off upper seven bits
            store the result in the CARRY location.
```

2. Specify the input and output required.

input required	- two 8-bit numbers
output required	- an 8-bit result and a 8-bit carry in memory.

3. Study the instruction set carefully. This step helps in specifying the available instructions and their format and constraints. For example, the segment registers cannot be directly initialized by a memory variable. Instead we have to first move the offset for segment into a register, and then move the contents of register to the segment register.

You can exit to DOS, by using interrupt routine 21h, with function 4Ch, placed in AH register.

It is a nice practice to first code your program on paper, and use comments liberally. This makes programming easier, and also helps you understand your program later. Please note that the number of comments do not affect the size of the program.

After the program development, you may assemble it using an assembler and correct it for errors, finally creating exe file for execution.

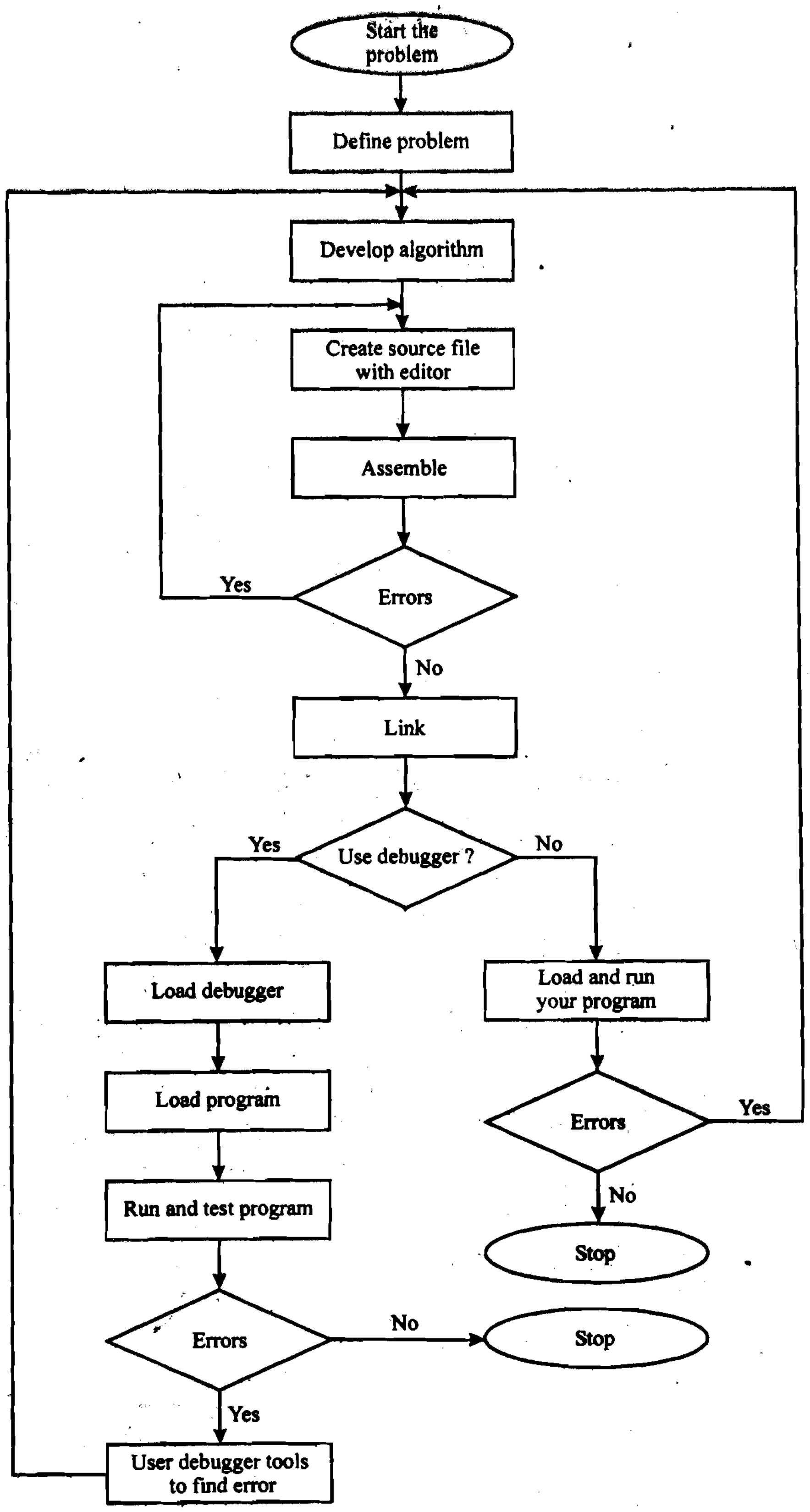
Check Your Progress 2

State True or False

		T	F
1.	For input/ output on Intel 8086/8088 machine running on DOS require special routines to be written by the assembly programmers.		
2.	Intel 8086 processor recognises only the software interrupts.		
3.	INT instruction in effect calls a subroutine, which is identified by a number.		
4.	Interrupt vector table IVT stores the interrupt handling programs.		
5.	INT 21H is a DOS function call.		
6.	INT 21H will output a character on the monitor if AH register contains 02.		
7.	String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively.		
8.	To perform final exit to DOS we must use function 4CH with the INT 21H.		
9.	Notepad is an editor package.		
10.	Linking is required to link several segments of a single assembly program.		
11.	Debugger helps in removing the syntax errors of a program.		
12.	COM program is loaded at the 0th location in the memory.		
13.	The size of COM program should not exceed 64K.		
14.	A COM program is longer than an EXE program.		
15.	STACK of a COM program is kept at the end of the occupied segment by the program.		
16.	EXE program contains a header module, which is used by DOS for calculating segment addresses.		
17.	EXE program cannot be easily debugged in comparison to COM programs.		
18.	EXE programs are more easily relocatable than COM programs.		

2.8 SUMMARY

We summarize the complete discussion in the following flow chart.



2.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. (a) It helps in better understanding of computer architecture and work in machine language.
   (b) Results in smaller machine level code, thus result in efficient execution of programs.
   (c) Flexibility of use as very few restrictions exist.

2. A segment identifier a group of instructions or data value. We have four segments.
   1. Data segment 2. Code segment 3. Stack segment 4. Extra Segment

3. (a) False
   (b) False
   (c) True
   (d) True
   (e) False
   (f) True

Check Your Progress 2

1. False
2. False
3. True
4. False
5. True
6. True
7. True
8. True
9. True
10. False
11. False
12. False
13. True
14. False
15. True
16. True
17. False
18. True

2.10 FURTHER READINGS

1. Yu-Cheng Lin, Genn. A. Gibson, "*Microcomputer System the 8086/8088 Family*" 2nd Edition, PHI.
2. Peter Abel, "*IBM PC Assembly Language and Programming*", 5th Edition, PHI.
3. Douglas, V. Hall, "*Microprocessors and Interfacing*", 2nd edition, Tata McGraw-Hill Edition.
4. Richard Tropper, "*Assembly Programming 8086*", Tata McGraw-Hill Edition.
5. M. Rafiquzzaman, "*Microprocessors, Theory and Applications: Intel and Motorala*", PHI.