UNIT 14 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Structure **Page No.**

14.0 INTRODUCTION

In the previous unit, you have gone through the basic concepts of 8086 microprocessor, which included the 8086 structure, segmentation, register set, instructions and addressing modes. This unit present a basic framework for writing assembly language programs for 8086 microprocessor. In this unit, you will learn about the importance, basic components and development tools of assembly language programming. The Input/Output to an assembly language program is a complex process. This unit discusses the Input/Output to assembly program by using interrupts. This unit also discussed about different kinds of Assembly programs, viz. COM programs and EXE programs. Finally,the unit presents an example assembly program. An assembly program consists of assembler directives and instructions of 8086 microprocessor. This program is assembled using an assembler program. Several such assembler programs exist, which use different assembler directives. We have used the assembler directives, as used in Microsoft Assembler (MASM). However, these directives may be different for different assemblers. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using and change directives accordingly.

14.1 OBJECTIVES

After going through this unit, you should be able to:

- define the need and importance of an assembly program;
- usebasic directives for writing an assembly program;
- use interrupts to perform input/ output in an assembly program;
- define and differentiate between COM and EXE programs.

14.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

The computer instructions are a sequence of 0's and 1's. These sequencesconsist of instruction operation code, addressing modes and operand addresses. The instructions of the programs written in the machine language are directly decoded by processing unit. However, you may have to face the following problems, if youprogram using machine language:

- Machine Language depends on machine instruction set and is difficult for most people to write in 0-1 forms.
- Debugging or correcting a machine language program is difficult.
- Deciphering the machine code is very difficult. Thus, program logic of programs written in machine language will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instructionsof 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 better 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 execution module. An assembly instruction is directing translated to a machine instruction, therefore, assembly programs are highly optimized. 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 the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line code for a single instruction. Further, complex instructions of a computer, like string instruction of 8086 highly that are highly optimized, can be used while writing an assembly program, making program faster. On the other hand, unlike high level languages,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 to write good logic of a program

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 High level languages with some highly efficient but non–portable routines. It will be worth mentioning here that UNIX is mostly written in C, a high-level language, but it has

about 5-10% machine dependent assembly code. In addition, many telecommunications applicationsuse assembly routinesto enhancethe efficiency.

14.3 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 anassembler program is used to translate the assembly language program into machine code. 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 (refer to Figure 14.1).

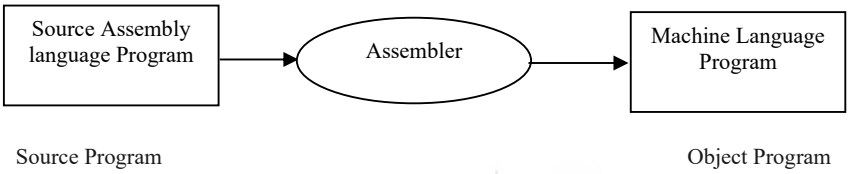

Figure 14.1: Use of assembler

Subsequently the object program is linked and an executable program is created. These steps are explained below:

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 generates a header in front of the.OBJ module; part of the header contains information about incomplete addresses in the object module. The .OBJ module is not an executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linkercompletes any address left open by the assembler and combines separately assembled programs into one executable module. In addition, it also initializes the .EXE module with special instructions to facilitate its subsequent loading of the .EXE program into the computer memory for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to be loaded in the memory, it is able to resolve all the remaining incomplete addresses in the header. The loader drops the header and creates a program segment prefix just before the program is loaded in memory.

These steps are shown in Figure 14.2

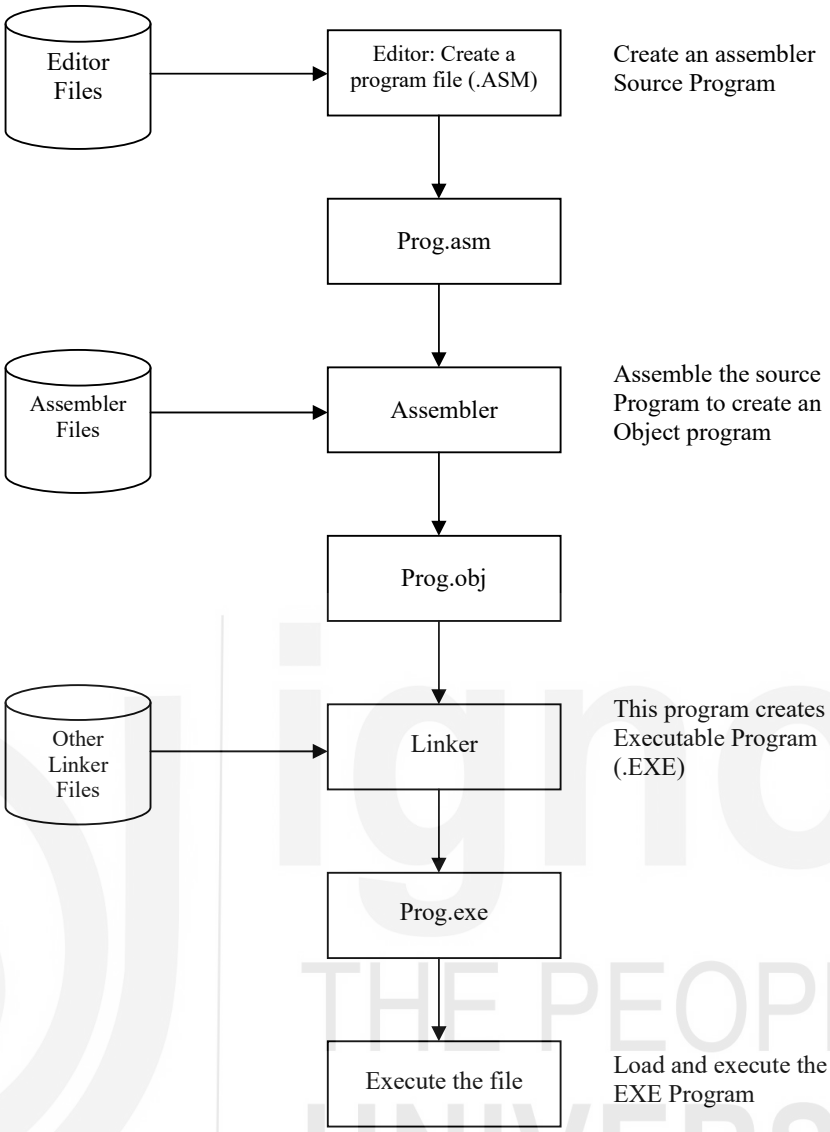

Figure 14.2: Program Assembly

Tools required for assembly language programming: Following are some of the basic tools needed to create assembly program. A modern-day assembler may contain several of these tools.

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 program createsan ASCII file. A common line editor program is NOTEPAD in Windows; vi editor in UNIX etc. An editor program may be part of assembler itself. You should use proper syntax of the assembly instructions to create an 8086-assembly program.

Assembler: An assembly program consists of assembly language instructions, which consists of assembly language mnemonics. The editor program, as defined above, is used to input the assembly language program and save it as a text file. An assembler is a program that converts the assembly language program, stored in a text file, into an equivalent machine language program. In general, this conversion is performed in two phases: first the assembler reads the assembly language file to collect various symbols used by the program along with their offsets in symbol table. On the second pass, it produces binary code for each instruction of the program and assignsan offset to all the symbols in the symbol table with respect to the segment base.

The assembler generates three files when your program gets successfully assembled with no errors. These three files are the object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program.The errors that are detected by the assembler are called the symbol errors. For example, in the following statement the mnemonic MOVE is compared by assembler to all the mnemonics of the mnemonic set. It fails to get a match following which it assumes MOVE to be an identifier and looksfor its entry in thesymbol table.It does not find it there too, therefore gives an error "undeclared identifier".

```
MOVE AX1, ZX1 ;
```

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 historical assemblers available on PCs are MASM, TURBO assembler etc.

Linker: For better modularity programs are broken into several sub routines. It is even better to design common routine, like reading a hexadecimal number, writing hexadecimal number etc., which could be used by a lot of other programs.These common routines can be put into files and 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 linked file, which contains the binary code for all component modules. The linker also produces link map, which contains the address information about the linked files. The linker, however, does not assign absolute or physical addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. Since these programs uses just relative addresses, they can be loaded in any physical memory address. Thus, these programs are called relocatable programs.

Loader: The basic purpose of the loader program is to convert the logical or relative addresses assigned by linker to absolute or physical memory addresses. This task is performed, while loader loads the linked program into the physical memory for execution. The linked program is brought from the secondary memory, like disk, to the computermemory for execution The file name extension of the files 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 contents of the register after the execution.
- It traces the execution of the specified segment of the program and displays the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., converts 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, Emulator of 8086, or any other assembler
- Linker, which may be included in the assembler
- Debugger for debugging, if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

a. Programming errors: They are the errors you can encounter in the course of executing a program written in any high-levellanguage, like syntax errors and semantic error
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. Therefore, assembly program should be tested in a safe mode.

14.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

In this section, a simple assembly program is shown and its various components are explained. Consider the following program:

```
Line    Offset
Numbers          Source Code
0001             DATA SEGMENT
0002    0000     MESSAGE DB "Assembly LanguageProgramming$"
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      MachineCode    Assembly Instructions
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
```

Program1: A simple 8086 assembly language program

The details of this program are explained in section 14.4.1.

14.4.1 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 names: Segment name is specified by the programmer. It allows programs written in 8086 assembly language to be relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Assembly Language Programming$" somewhere in memory. It is located in the DATA SEGMENT. Since the characters are stored in ASCII, therefore it will occupy 30 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 astatement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a new line character.

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 the name of an item,given by you, 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

   the statement
   A20: BL,45 ; defines a label A20.

Identifier can use alphabet, digit or special character.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.

14.4.2 Directives

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

1. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. This statement directs the assembler to treat related tokens in the source file as numeric constants in hexadecimal notation.

2. **PROC Directive:**PROC directive can be used in the code segment of an assembly program to create a procedure. You can use more than one PROC directives in a code segment. A PROC directive marks the start of a procedure. The end of a procedure is marked by the ENDP directive. The following example shows the start and end of a procedure named CheckZero.

```
CheckZero PROC  NEAR  ; Beginning of a Procedure in same segment
   …
CheckZero ENDP  NEAR  ; Marks the end of the Procedure CheckZero
```

3. **END DIRECTIVE:** There are three different END directives. These are:
   (i) ENDS Directive: This directive marks the completion of a segment. Thus, every segment used by you must have an ENDS directive.
   (ii) ENDP directive:As stated in point 2 it is used to mark the end of a procedure.
   (iii) END directive: It marks the end of the entire program. Any statement after this directive is ignored by the assembler.

4. **ASSUME Directive:** The purpose of assume directive is to identify the possible use of various segments defined in an assembly program. For example, if in your assembly program you have defined segments named STRING, CODE and STACK, which are to be used as data segment, code segment and stack segment respectively, then you can use the following ASSUME directive statement

   ```
   ASSUME  CS: CODE, DS: STRING; SS: STACK
   ```

5. **SEGMENT Directive:** The segment directive defines the segment name. A segment directive makes it possible to set a segment register to address the base address of a segment register (Please refer to discussion on segment register in Unit 13). All the offsets in a segment are computed from a base address of a segment.A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

CODE SEGMENT

The code segment contains the code of the program, which may include procedures and sometimes other segments too. Linker marks the code segment in a program in a header. This header is used by the operating system when it invokes the loader to load an executable file of the program into memory. The loader reads this header for setting the CS register. A physical memory address is represented as CS: xxxx, where xxxx represents the offset in the code segment. In general, the first instruction of the code segment is assumed as the first instruction to be executed, therefore, is put at an offset of 0000H. The instruction pointer (IP) register is used to mark the offset of an instruction in code segment. The CS: IP pair is thus used to specify physical address of an instruction in a program that is being executed.

STACK SEGMENT

8086Microprocessor 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 issues a real time clock interrupts after every 55 milliseconds. 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 before the occurrence of interrupt. All such information gets recorded in the stack. Please note if you have not specified the stack segment it is automatically created.Why is stack segment essential? Consider your program is being executed by CPU, and a clock pulse need service, then if the system has no stack, then your CPU will not be able to return to your program again after serving of the clock pulse.

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: Specifies the data type to assemble.
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; i.e. 10 zero values.

EQU directive is used to define a name to a constant

```
CONST    EQU    20
```

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

Some other examples of using these directives are:

BINDB0111001B	; Binary value in byte operand named temp
OCT DW 7341Q	; Octal value assigned to word variable named VALI
DECDB49	; Decimal value 49 contained in byte variable names DEC
HEXDW03B2AH	; Hexadecimal value a is stored in a word operand HEX.

ASCDB 'EXAMPLE'	; Array of ASCII values is stored in variable ASCI.

☞ 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.	☐	

14.5 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 performed using these interrupts.

14.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are caused by devices like keyboard, printer, monitor, an error condition, etc.

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

Hardware interrupts are generated by a device that requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

In 8086 software interrupts can be used for input-output of data. A software interrupt is initiated using the following statements:

```
INT     number
```

In 8086, this interrupt instruction is processing using an**interrupt vector table (IVT).** The IVT is located in the first 1K bytes of memory, and has a total of 256 entries,

each of 4 bytes. 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 14.3 shows the processing of an interrupt.

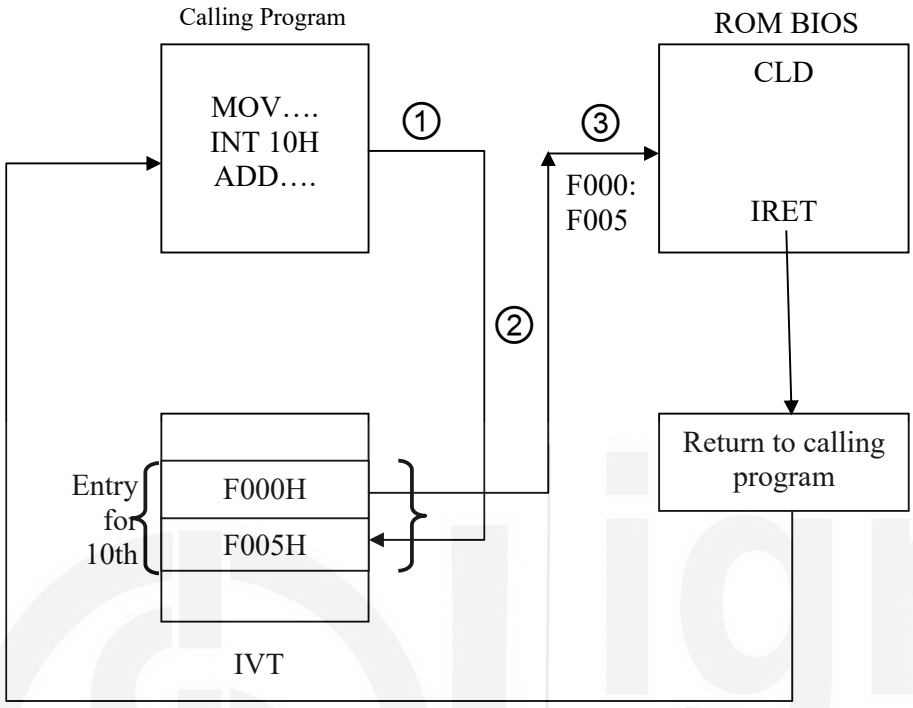

Figure 14.3: Processing of an Interrupt

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 10Hwould be found at IVT at an address 40H. Similarly, the entry of INT 3H will be at an address 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 address (CS: IP) as F000h:F065h

Step 3: The CPU loads the CS register and the IP register, with this new address in the IVT, and starts instruction execution process for that instruction.

Step 4: IRET (interrupt return) causes the program to resume execution of the next instruction of the program, which was being executed prior to interrupt servicing.

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 a character is a difficult program. However, these tasks were simplified by the architecture of the 8086. This architecture provides a pack of software interrupt vectors beginning at address 0000h:0000h,i.e., start of IVT.

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

14.5.2 DOS Function Calls (Using INT 21H)

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

Some important DOS function calls are given in the following table:

DOS Function Call	Purpose and Example
AH = 01H	This function called is used for reading a single character from keyboard and displaying it on monitor. The input value is put in AL register. For example, to read a character in a memory location X, you may use the following code fragment: `MOV AH,01H ; load AH register with the function value 01h` `INT 21H;` call the interrupt to read a character in AL `MOV X, AL;` Load the read character in memory location X

AH = 02H	This function prints an 8-bit data (normally ASCII), which is stored in DL register, on the screen. For example, to print a character '?' on the monitor, you may write the following code fragment:
	`MOV AH,02H ; load AH register with the function value 02h`
	`MOV DL, '?' ; Move the character to be displayed in DL`
	`INT 21H;` call the interrupt to display the character in DL

AH = 08H	This function is also used to input a single character into AL register, except that the character does not get displayed on the monitor. For example, to read a character in a memory location X, you may use the following code fragment: `MOV AH,08H ; load AH register with the function value 08h` `INT 21H;` call the interrupt to read a character in AL `MOV X, AL;` Load the read character in memory location X

AH = 09H	This function outputs a string whose offset is stored in DX register. The string is terminated by using a $ character. You can use this function to print newline character, tab character etc. For example, to print a string "Hello World", you may use the following code fragment:
	`DATA SEGMENT`
	`STRING DB 'HELLO WORLD','$'`
	`DATA ENDS`
	`CODE SEGMENT`
	…

<table>
<tr><td></td><td>

```
MOV AX, DATA; Put offset of Data Segment to AX.
  MOV DS, AX; Initialize data segment register using AX
  MOV AH,09H; load AH with the function value 09h
  MOV DX, OFFSET STRING; Store the offset of STRING in DX
INT 21H    ; Call interrupt 21H to display the STRING
  …
```

</td></tr>
<tr><td>AH = 0AH</td><td>
For input of string up to 255 characters. The string is stored in a buffer. For example, the following data and code fragment will input a string having a maximum length of 50 bytes. First, you need to define these parameters in the data segment, as given below:

```
DATA SEGMENT
     BUFFDB50
     DB?
     DB 50 DUP(0)
DATA ENDS
```

The name of the data segment, as given above, is DATA. It consists of total 52 bytes locations named BUFF. The first location of BUFF stores the decimal value 50, which is the maximum size of the string that can be stored in this buffer. The second location, marked with '?', will be used to store the actual size of the string, once it is read in the buffer. The remining 50 bytes at present are initialized as 0. These bytes will contain the string once it is read. The code segment, which will perform the string read operation is given below:

```
CODE SEGMENT
     …
     MOV AH, 0AH    ; Move 0A to AH register
     MOV DX, OFFSET BUFF; DX contains offset of BUFF
     INT   21H          ; Call interrupt 21h
      …
CODE ENDS
```

For the given code (complete the other necessary directives and statements) and data segment, if you input a value "Parv", then it will be stored in the BUFF as given below:

50	4	P	a	r	v	0	0	…	0

</td></tr>
<tr><td>AH = 4CH</td><td>This function call returns control back to the operating system.</td></tr>
</table>

Some examples of Input

(i) Input a single ASCII character to BL register, without displaying it on screen

```
CODE SEGMENT
  :
  MOV   AH, 08H   ; AH is loaded with function 08H
```

```
  INT   21H        ;Issue the Interrupt 21h
  MOV   BL, AL     ;  Transfer the input obtained in AL to BL
  :
CODE ENDS
```

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

```
CODE SEGMENT
  :
  MOV   AH,  01H
  INT   21H
  MOV   BL, AL
  SUB   BL,'0'   ; or use SUBBL,30H
  :
CODE ENDS.
```

Description: First you move the value of function 01H to AH register, next you call the Interrupt 21H. Execution of these two statements will result in input of a single character in AL register and display of that character on the monitor. You expect this character to be any digit amongst 0 to 9 (you can check this with additional assembly code, if needed.). The ASCII equivalent values of these digits '0' to digit '9' are 30H to 39H. The input value first is moved from AL to BL register and thereafter you either subtract '0', which is ASCII value of digit 0. Alternatively, can subtract 30H from BL, which is same as the hexadecimal value of digit '0'. This subtraction will result in binary equivalent value of the input digit in the BL register. For example, if you had input digit '8' as input, then BL register will contain: 38H-30H= 08H. This binary value then can be used for arithmetic operations.

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

One of the methods to input two-digit number would be to input two single digit numbers and using the place value of these digits convert them to a two-digit binary equivalent number. For example, to input a two-digit number 48, you will first input digit 4, convert it to binary and then input digit 8, which is also converted to binary. Now, perform the following computation to get an equivalent binary number into the specific register:

4*10+8 = 48

The assembly code for the same is given below:

```
CODE SEGMENT
  :
  MOV   AH,08H     ; Function 08H
  INT   21H        ; Interrupt 21H
  MOV   BL, AL     ; Move the value to BL register
  SUB   BL,'0'     ; Subtract '0' to get binary in BL
  MOV   AH, 08H    ; Now, start input of second digit
  INT   21H        ;
  MOV   DL, AL     ; Store AL in DL register
  SUB   DL, '0'    ; get the second binary digit in DL
  MOV   AL, 0AH    ; Move value 10 in AL register
  MUL   BL         ; Multiply AL by BL.
  ADD   BL, DL
  :
CODE ENDS.
```

Description: The code first input the two digits in BL and DL registers respectively, for example, an input 4 will be moved to BL and 8 will be moved to DL register, where they are converted to equivalent binary using subtraction. Next, BL is multiplied by 10, which is moved to AL register and added with the

value of DL, resulting in 4*10+8. This output is stored as binary value in BL register.

Examples of output on Display unit

(i) **Displaying a single character**

The following code displays the ASCII equivalent character of the data stored in BL register.

```
CODE SEGMENT
  :
  MOV   AH,02H      ; The single character output function
  MOV   DL, BL      ; Move the character to DL
  INT   21H
  :
CODE ENDS.
```

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

The following program first convers the binary value to equivalent decimal digit and then outputs it on the monitor. For example, if BL register contains a value '0000 0111', which is 07H, it will be converted to digit 7 by adding 30H or '0' and moved to DL register so that interrupt instruction displays this value on the monitor. The following code will perform the display as stated above:

```
CODE SEGMENT
  :
  ADD BL, '0'
  MOV DL, BL
  MOV AH,02H
  INT   21H
  :
CODE ENDS.
```

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

Assuming that the two-digit number 48 is stored as number 4 in BH and number 8 in BL. Each of these digits is converted to its equivalent ASCII digit by adding 30H or '0'. This is followed by displaying the equivalent ASCII of the digits respectively to output the given number.

```
CODE SEGMENT
  :
  ADD   BH, '0'
  ADD   BL, '0'
  MOV   AH, 02H
  MOV   DL, BH
  INT   21H
  MOV   DL, BH
  INT   21H
  :
  CODE ENDS
```

(iv) **Displaying a string**

Assume that a string is stored in the data segment with the label ST1. To display this string the following code can be used:

```
DATA SEGMENT
  ST1 DB "Output a string$"
```

```
DATA ENDS
CODE SEGMENT
      :
      MOV DX, OFFSET ST1
      MOV AH, 09H
       INT   21H
      :
    CODE ENDS.
```

An example with Input and output both:

To write a program that accepts an input character from the keyboard and respond. "The input is _".

```
DATA SEGMENT
      MESSAGE DB "The input is$"
DATA ENDS
CODE SEGMENT
    MOV  AX, DATA; Move data segment address to AX
    MOV DS,  AX  ; Initialize DS register
    MOV  AH,  08H ; Set function for character read
    INT  21H      ; Read character in AL
    MOV  BL, AL   ; Move input to BL
    MOV AH, 09H   ; Function to display strings
    MOV DX, OFFSET MESSAGE ; Move offset of string to DX
    INT  21 H         ; Display string named MESSAGE
    MOV  AH,  02H; Function to display character
    MOV DL, BL        ; Move character to DL
    INT  21 H         ; Display character
    MOV AX, 4C00H     ; Move 4CH to AH (DOS function call)
    INT 21H           ; Exit to DOS
CODE ENDS
END.
```

☞ Check Your Progress 2:

Q1: List the interrupts that can be used to input one character.

.....

.....

Q2: What is the output of following code segment, assume that BL register contains the binary value $0000\ 0010_2$

```
CODE SEGMENT
      :
      ADD BL, '0'
      MOV DL, BL
       MOV AH,02H
       INT   21H
      :
    CODE ENDS.
```

.....

.....

Q3: Name the interrupt used to exit to operation to operating system.

.....

.....

14.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

- COM Programs: A .COM program has only one physical segment, which includes all the different segments.
- EXE Program: An EXE programconsists of separate segments.

Let us look into brief details of these programs.

14.6.1 COM Programs

A COM (Command) program is the binary image of a machine language program. It is loaded in the memory at the lowestavailable segment address. In 8086 micro-processor, a COM program code begins at an offset 100h, as the first 1K locations are occupied by the interrupt vector table (IVT).

All the segments of a COM program are kept in the same segment, i.e., its code segment, data segment and stack segments are within the same segment. Since the offsets in a physical segment can be of size 16 bits, therefore the size of COM program is limited to 2^{16} = 64K which includes code segment, data segment and stack segment.The following program is a COM program, which adds two numbers. The program stores the result of addition and carry bit of addition in memory variables.

```
    CSEG SEGMENT
         ASSUME CS:CSEG, DS:CSEG, SS:CSEG
         ORG 100h      ;  Segment starts at address 0100h
START:   MOV AX, CSEG ;Move the segment address to AX
         MOV DS, AX    ;  Initialize Data segment using AX
         MOV AL, NUM1  ;  Transfer first operand to AL
         ADD AL, NUM2  ;  Add second operand to AL
         MOV RES, AL   ;  Store the result in AL to location RES
         RCL AL, 01    ;  Rotate AL by 1 bit to get carry into LSB
         AND AL, 00000001B;  Mask out all bits except the LSB
         MOV CARRY, AL      ;  Store the carry bit into location CARRY
         MOV AX,4C00h
         INT 21h
         NUM1DB   15h  ;  The first operand
         NUM2 DB  20h  ;  The Second operand
         RES DB   ?    ;  Stores the sum
         CARRY DB ?    ;  Stores the carry bit
   CSEG ENDS
   END START
```

The program initializes the data segment CSEG, which is also the code and stack segment too. The two operands are stored in memory locations NUM1 and NUM2 are added and the result is stored in the location RES. In order to store the carry bit first a rotate with carry instruction is executed, followed my masking out the upper 7 bits. This causes only the carry bit to remain in the AL register. This carry bit is then moved to the location CARRY. Finally, the program exits to DOS using Interrupt 21H.

The COM programs are stored on a disk with an extension ***.com***. A COM program uses less disk space in comparison to an 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.

14.6.2 EXE Programs

An EXE program is stored on the disk with extension ***.exe***. EXE programs are longer than the equivalent 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 compute the address of various segments and other components related to offsets. A detailed discussion on segment header is beyond the scope of this Unit.

The load module of EXE program consists of several segments of length up to 64K. It may be noted that in 8086 microprocessor a maximum of four segments may be active at any time. These segments can be of variable sizes, with the maximum size being 64K.

In the subsequent Units, you will be learning to write EXE programs only as:

- EXE programs are better suited for debugging.
- Assembled EXE programs can be easily linked to subroutines of high-level languages.
- EXE programs are easily to relocate in the memory, as they do not contain any ORG statement. It may be noted that ORG statement forces a program to be loaded from a specific memory 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 EXE program, which is equivalent to the COM program given in the previous section is given below:

```
        DATA SEGMENT
            NUM1DB   15h  ;  The first operand
            NUM2 DB  20h  ;  The Second operand
            RES DB   ?    ;  Stores the sum
            CARRY DB ?    ;  Stores the carry bit
        DATA END
        CODE SEGMENT
            ASSUME CS:CODE, DS:DATA
START:      MOV AX, DATA ;Move the segment address to AX
            MOV DS, AX    ;  Initialize Data segment using AX
            MOV AL, NUM1 ;  Transfer first operand to AL
            ADD AL, NUM2 ;  Add second operand to AL
            MOV RES, AL   ;  Store the result in AL to location RES
            RCL AL, 01    ;  Rotate AL by 1 bit to get carry into LSB
            AND AL, 00000001B;  Mask out all bits except the LSB
            MOV CARRY, AL     ;  Store the carry bit into location CARRY
            MOV AX,4C00h
            INT 21h
    CODE ENDS
    END START
```

14.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

This section defines the art of writing good assembly programs. A good program requires a clear description or problem and the algorithm that is being used for solving the problem. The following are some of the advices, which may help you in writing good assembly programs.

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:
   - Assuming that both the numbers, NUM1 and NUM2 are in the memory.
     - Put first number from memory to AL
     - Add second number from memory to AL
     - Store the result in some memory location
   - Position carry bit in Least significant bit (LSB) of a byte
     - mask off upper seven bits
     - store the result in the CARRY location.

2. Specify the input and output of the program.

   Input: Two 8-bit numbers, in two different memory locations
   Output: An 8-bit result and an 8-bit carry in memory locations

3. Study the instruction set carefully: Study the available set of instructions, their format and their limitations. For example, the limitation of the move instruction is that it cannot move an immediate operand to a segment register. Thus, the segment address is first moved to a register, say AX, which is then used to initialize the segment register. cannot be directly initialized by a memory variable.
4. You can exit to DOS, by using interrupt routine 21h, function 4Ch. Therefore,04CH is placed in AH register followed by INT 21H instruction. This will result in exit to DOS.
5. 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 does not affect the size of your program.
6. You may assemble your program using an assembler, which helps you in removing the syntax errors. It also helps in creating an ***.exe*** file for execution.

☞ Check Your Progress 3

Q1: When would you use ***.com*** program?

.....

.....

.....

.....

Q2: Why are EXE programs preferred over .COM programs?

.....

.....

.....

.....

Q3: State True or False

	T	F
(i) Input/output on Intel 8086/8088 machine running on DOS require special functions to be written by the assembly programmers.	☐	
(ii) Intel 8086 processor recognizes only the software interrupts.	☐	

(iii) INT instruction in effect calls a subroutine, which is identified by a number. ☐

(iv) Interrupt vector table IVT stores the interrupt handling programs. ☐

(v) INT 21H is a DOS function call. ☐

(vi) INT 21H will output a character on the monitor if AH register contains 02. ☐

(vii) String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively. ☐

(viii) To perform final exit to DOS we must use function 4CH with the INT 21H. ☐

(ix) Notepad can be used as an editor package. ☐

(x) Linking is required to link several segments of a single assembly program. ☐

(xi) Debugger helps in removing the syntax errors of a program. ☐

(xii) COM program is loaded at the 0^{th} location in the memory. ☐

(xiii) The size of COM program should not exceed 64K. ☐

(xiv) A COM program is longer than an EXE program. ☐

(xv) STACK of a COM program is kept at the end of the occupied segment by the program. ☐

(xvi) EXE program contains a header module, which is used by DOS for calculating segment addresses. ☐

(xvii) EXE program cannot be easily debugged in comparison to COM programs. ☐

(xviii) EXE programs are more easily relocatable than COM programs. ☐

14.8 SUMMARY

This unit introduces you to some of the basic concept of 8086 programming,especially input/output. 8086 microprocessor uses an interrupt vector table (IVT) that points to the address of the interrupt servicing programs of 8086 micro-processor. One of the most important interrupts being interrupt 21H, which is used for input/output and several different functions. An IVT provides a flexible design environment, as you can change the interrupt service program without much efforts. This unit discusses some of the important functions of INT 21H. This unit also differentiatesbetween COM & EXE program that are used in 8086 micro-processor.

14.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. (a) It helps in better understanding of computer architecture and 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 identifies 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

Q1: Interrupt 21H with functions 01H, 08H and 0AH
Q2: Output will be the digit 2.
Q3: Interrupt 21H with function 4C

Check Your Progress 3

Q1: The COM programs are of size less than 64K. When you require small fast functions, you may use COM programs.
Q2: Due to better structure and possibility of linking with the high level language programs. In addition, if your environment supports multiprogramming then EXE programs can be easily relocated
Q3:

(i) False
(ii) False
(iii) True
(iv) False
(v) True
(vi) True
(vii) True
(viii) True
(ix) True
(x) False
(xi) False
(xii) False
(xiii) True
(xiv) False
(xv) True
(xvi) True
(xvii) False
(xviii) True

14.10 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.