Table 3.1 (*Contd.*)

COMMAND CHARACTER	*Format/Formats*	*Functions*
`-s`	`SEG:OFFSET1 to OFFSET2 BYTE/BYTES <ENTER>`	Searches a BYTE or string of BYTES separated by ',' in the memory block SEG:OFFSET1 to OFFSET2, and displays all the offsets at which the byte or string of bytes is found.
`-q`	`<ENTER>`	Quit the DEBUG and return to DOS
`-T`	`SEG:OFFSET <ENTER>`	Trace the program execution by single stepping starting from the address SEG:OFFSET.
`-m`	`SEG:OFFSET1 OFFSET2 NB <ENTER>`	Move NB bytes from OFFSET1 to OFF-SET2 in segment SEG
`-c`	`SEG:OFFSET OFFSET2 NB <ENTER>`	Copy NB bytes from OFFSET1 to OFF-SET2 in segment SEG.
`-n`	`FILENAME.EXE <ENTER>`	Set filename pointer to FILENAME
`-l`	`<ENTER>`	Load the file FILENAME.EXE as set by the -n command in the RAM and set the CS:IP at the address at which the file is loaded.

- Note that, changing the case of the command letters does not change the command option.
- The entered numbers are considered as hexadecimal.

3.4 ASSEMBLY LANGUAGE EXAMPLE PROGRAMS

In the previous chapter, we studied the complete instruction set of 8086/88, the assembler directives and pseudo-ops. In the previous sections, the procedure of entering an assembly language program into a computer and coding it, i.e. preparing an EXE file from the source code were described. In this section, we will study some programs which elucidate the use of instructions, directives and some other facilities that are available in assembly language programming.

After studying these programs, it is expected that the reader would have got a clear idea about the use of different directives, pseudo-ops and their syntaxes, besides understanding the logic of each program. If one writes an assembly language program and tries to code it, the chances of error are high in the first attempt. Error free programming depends upon the skill of the programmer, which can be developed by writing and executing a number of assembly programs, besides studying the example programs given in this text.

Before explaining the written programs, we have to explain an important point about the DOS function calls available under INT 21H instruction. DOS is an operating system, which is a program that stands between a bare computer system hardware and a user. It acts as a user interface with the available computer hardware resources. It also manages the hardware resources of the computer system. In the Disk Operating System, the hardware resources of the computer system like memory, keyboard, CRT display, hard disk, and floppy disk drives can be handled with the help of the instruction INT 21H. The routines required to refer to these resources are written as interrupt service routines for 21H interrupt. Under this interrupt, the specific resource is selected depending upon the value in AH register. For example, if AH contains 09H, then CRT display is to be used for displaying a message or if, AH contain 0AH, then the keyboard is to be accessed. These interrupts are called 'function calls' and the value in AH is called 'function value'. In short, the purpose of 'function calls' under INT 21 H is to be decided by the 'function value' that is in AH. Some function values also control the software operations of the machine. The list of the function values available under INT 21 H, their corresponding functions, the required parameters and the returns are given in tabulated form in the Appendix-B. Note that there are a number of interrupt functions in DOS, but INT 21H is used more frequently. The readers may find other interrupts of DOS and BIOS from the respective technical references.

In this chapter, a few example programs are presented. Starting from very simple programs, the chapter concludes with more complex programs.

Program 3.1

Write a program for addition of two numbers.

Solution The following program adds two 16-bit operands. There are various methods of specifying operands depending upon the addressing modes that the programmer wants to use. Accordingly, there may be different program listings to achieve a single programming goal. A skilled programmer uses a simple logic and implements it by using a minimum number of instructions. Let us now try to explain the following program:

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
OPR1 DW 1234H                      ; 1st operand
OPR2     DW 0002H                  ; 2nd operand
RESULT   DW 01 DUP(?)              ; A word of memory reserved for re-
                                     sult
DATA     ENDS
CODE     SEGMENT
START:   MOV AX, DATA              ; Initialize data segment
         MOV DS, AX                ;
         MOV AX, OPR1              ; Take 1st operand in AX
         MOV BX, OPR2              ; Take 2nd operand in BX
         CLC                       ; Clear previous carry if any
         ADD AX, BX                ; Add BX to AX
         MOV DI, OFFSET RESULT     ; Take offset of RESULT in DI
         MOV [DI], AX              ; Store the result at memory address in DI
         MOV AH, 4CH               ; Return to DOS prompt
         INT 21H
CODE     ENDS                      ; CODE segment ends
         END START                 ; Program ends
```

Program 3.1(a) *Listings*

3.4.1 How to Write an Assembly Language Program

The first step in writing an assembly language program is to define and study the problem. Then, decide the logical modules required for the program. From the statement of the program one may guess that some data may be required for the program or the result of the program that is to be stored in memory. Hence the program will need a logical space called DATA segment. Invariably the CODE segment is a part of a program containing the actual instruction sequence to be executed. If the stack facility is to be used in the program, it will require the STACK segment. The EXTRA segment may be used as an additional destination data segment. Note that the use of all these logical segments is not compulsory except for the CODE segment. Some programs may require DATA and CODE segments, while the others may also contain STACK and EXTRA. For example, Program 3.1 (a) requires only DATA and CODE segment.

The first line of the program containing the 'ASSUME' directive declares that the label CODE is to be used as a logical name for CODE segment and the label DATA is to be used for DATA segment. These labels CODE and DATA are reserved by MASM for these purposes only. They should not be used as general labels. Once this statement is written in the program, CODE refers to the code segment and DATA refers to data segment throughout the program. If you want to change it in a program you will have to write another ASSUME statement in the program.

The second statement, DATA SEGMENT marks the starting of a logical data space DATA. Inside the DATA segment, OPR1 is the first operand. The directive DW defines OPR1 as a word operand of value 1234H and OPR2 as a word operand of value 0002H. The third DW directive reserves 01H words of memory for storing the result of the program and leaves it undefined due to the directive DUP(?). The statement DATA ENDS marks the end of the DATA segment. Thus the logical space DATA contains OPR1, OPR2 and RESULT, which will be allotted physical memory locations whenever the logical SEGMENT DATA is allocated memory or loaded in the memory of a computer as explained in the previous topic of relocation. The assembler calculates that the above data segment requires 6 bytes, i.e. 2 bytes each for OPR1, OPR2 and RESULT.

The code segment in the above program starts with the statement CODE SEGMENT. The code segment, as already explained, is a logical segment space containing the instructions. The label STARTS marks the starting point of the execution sequence. The ASSUME statement just informs the assembler that the label CODE is used for the code segment and the label DATA is used for the DATA segment. It does not actually put the address corresponding to CODE in Code Segment (CS) register and address corresponding to DATA in the Data Segment (DS) register. This procedure of putting the actual segment address values into the corresponding segment registers is known as segment register initialisation. A programmer has to carry out these initializations for DS, SS and ES using instructions, while the CS is automatically initialised by the loader at the time of loading the EXE file into the memory for actual execution. The first two instructions in the program are for data segment initialization.

Note that, no segment register in 8086 can be loaded with immediate segment address value, instead the address value should be first loaded into any one of the general purpose registers which can then be transferred to any of the segment registers DS, ES and SS. Also one should note that CS cannot be loaded at all. Its contents can be changed by using a long jump instruction, a call instruction or an interrupt instruction. For each of the segments DS, ES and SS, the programmer will have to carry out initialization if they are used in the program, while CS is automatically initialized by the loader program at the time of loading and execution. Then the two instructions move the two operands OPR1 and OPR2 in AX and BX respectively. Carry is cleared before addition operation (optional in this program). The ADD instruction will add BX into AX and store the result in AX. The instruction used to store the result in RESULT uses a different addressing mode than that used for taking OPR1 into AX. The indexed addressing mode is used to store the result of addition in memory locations labeled RESULT.

The instruction MOV DI, OFFSET RESULT stores the offset of the label RESULT into DI register. The next instruction stores the result available in AX into the address pointed to by DI, i.e. address of the RESULT. A lot has been already discussed about the function calls under INT 21H. The function value 4CH is for returning to the DOS prompt. If instead of these one writes HLT instruction there will not be any difference in program execution except that the computer will hang as the processor goes to HLT state, and the user will not be able to examine the result. In that case, for further operation, one will have to reset the computer and boot it again. To avoid this resetting of the computer every time you run the program and enable it to check the result, it is better to use the function call 4CH at the end of each program so that after executing the program, the computer returns back to DOS prompt. The statement CODE ENDS marks the end of the CODE segment. The statement END START marks the end of the procedure that started with the label START. At the end of each file, the END statement is a must.

Until now, we have discussed Program 3.1(a) in significant detail. As we have already said, the program contains two logical segments CODE and DATA, but it is not at all necessary that all the programs must contain the two segments. A programmer may use a single segment to cover up data as well as instructions. Program 3.1(b) explains the fact.

```
ASSUME CS:CODE
         CODE        SEGMENT
         OPR1        DW     1234H
         OPR2        DW     0002H
         RESULT      DW     01 DUP(?)
START :  MOV AX, CODE
         MOV DS, AX
         MOV AX, OPR1
         MOV BX, OPR2
         CLC
         ADD AX, BX
         MOV DI, OFFSET RESULT
         MOV [DI], AX
         MOV AH,4CH
         INT 21H
CODE     ENDS
         END START
```

Program 3.1(b) *Alternative listing for Program 3.1*

We have discussed all the properties of this program in detail. For all the following programs, we will not explain the common things like forming segments using directives and operators, etc. Instead, just the logic of the program will be explained. The use of proper syntax of the 8086/8088 assembler MASM is self explanatory. The comments may help the reader in getting the ideas regarding the logic of the program.

Assemble the above written program using MASM after entering it into the computer using the procedure explained in Section 3.3.1. Once you get the EXE file as the output of the LINK program, Your listing is ready for execution. The Program 3.1 is prepared in the form of EXE file with the name KMB.EXE in the directory. Next, it can be executed with the command line as given below.

```
C> KMB
```

This method of execution will store the result of the program in memory but will not display it on the screen. To display the result on the screen the programmer will have to use DOS function calls, which will make the programs too lengthy. Hence, another method to check the results is to run the program under the control of DEBUG. To run the program under the control of debug and to observe the results one must prepare the LST file, that gives information about memory allotment to different labels and variables of the program while assembling it. The LST file can be displayed on the screen using NE-Norton's Editor.

Program 3.2

Write a program for the addition of a series of 8-bit numbers. The series contains 100(numbers).

Solution In the first program, we have implemented the addition of two numbers. In this program, we show the addition of 100 (D) numbers. Initially, the resulting sum of the first two numbers will be stored. To this sum, the third number will be added. This procedure will be repeated till all the numbers in the series are added. A conditional jump instruction will be used to implement the counter checking logic. The comments explain the purpose of each instruction.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT                            ; Data segment starts
NUMLIST DB 52H, 23H,-                   ; List of byte numbers
COUNT EQU 100D                          ; Number of bytes to be added
RESULT DW 01H DUP(?)                    ; One word is reserved for result
DATA ENDS                               ; Data segment ends
CODE SEGMENT                            ; Code segment starts at relative
ORG 200H                                ; address 0200h in code segment
START:       MOV AX, DATA               ; Initialize data segment
             MOV DS, AX
             MOV CX, COUNT              ; Number of bytes to be added in CX
             XOR AX, AX                 ; Clear AX and CF
             XOR BX, BX                 ; Clear BH for converting the byte to
                                          word
             MOV SI,OFFSET NUMLIST      ; Point to the first number in the
                                          list
AGAIN:       MOV BL, [SI]               ; Take the first number in BL,BH is zero
             ADD AX, BX                 ; Add AX with BX
             INC SI                     ; Increment pointer to the byte list
             DEC CX                     ; Decrement counter
             JNZ AGAIN                  ; If all numbers are added,point to re-
                                          sult
             MOV DI, OFFSET RESULT      ; destination and store it
             MOV [DI], AX
             MOV AH, 4CH                ; Return to DOS
             INT 21H
             CODE ENDS
END          START
```

Program 3.2 *Listings*

The use of statement ORG 200H in this program is not compulsory. We have used this statement here just to explain the way to use it. It will not affect the result at all. Whenever the program is loaded into the memory whatever is the address assigned for CODE, the executable code starts at the offset address 0200H due to the above statement. Similar to DW, the directive DB reserves space for the list of 8-bit numbers in the series. The procedure for entering the program, coding and execution has already been explained. The result of addition will be stored in the memory locations allotted to the label RESULT.

Program 3.3

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the locations starting from a known address.

Solution Compare the *i*th number of the series with the (*i*+1)th number using CMP instruction. It will set the flags appropriately, depending upon whether the ith number or the (*i*+1)th number is greater. If the *i*th number is greater than (*i*+1)th, leave it in AX (any register may be used). Otherwise, load the (*i*+1)th number in AX, replacing the *i*th number in AX. The procedure is repeated till all the members in the array have been compared.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT                          ; Data segment starts
LIST DB 52H, 23H, 56H, 45H, --        ; List of byte numbers
COUNT EQU OF                          ; Number of bytes in the list
LARGEST DB 01H DUP(?)                 ; One byte is reserved for the largest
                                        number.
DATA ENDS                             ; Data segment ends
CODE SEGMENT                          ; Code segment starts.
START:      MOV AX, DATA              ; Initialize data segment.
            MOV DS, AX
            MOV SI, OFFSET LIST
            MOV CL, COUNT             ; Number of bytes in CL.
            MOV AL, [SI]              ; Take the first number in AL
AGAIN:      CMP AL, [SI+1]            ; and compare it with the next number.
            JNL NEXT
            MOV AL, [SI+1]
NEXT:       INC SI                    ; Increment pointer to the byte list.
            DEC CL                    ; Decrement counter.
            JNZ AGAIN                 ; If all numbers are compared, point to
                                        result
            MOV SI, OFFSET LARGEST ; destination and store it.
            MOV [SI], AL
            MOV AH, 4CH               ; Return to DOS.
            INT 21H
            CODE ENDS
END         START
```

Program 3.3 *Listings*

Program 3.4

Modify the Program 3.3 for a series of words.

Solution The logic is similar to the previous program written for a series of byte numbers. The program is directly written as follows without any comment leaving it to the reader to find out the use of each instruction and directive used.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 1234H, 2354H, 0056H, 045AH, -
COUNT EQU OF
LARGEST DW 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV SI, OFFSET LIST
```

```
                MOV CL, COUNT
                MOV AX, [SI]
AGAIN:          CMP AX, [SI+2]
                JNL NEXT
                MOV AX, [SI+2]
NEXT:           INC SI
                INC SI
                DEC CL
                JNZ AGAIN
                MOV SI, OFFSET LARGEST
                MOV [SI], AX
                MOV AH, 4CH
                INT 21H
CODE            ENDS
                END      START
```

Program 3.4 *Listings*

Program 3.5

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Solution The simplest logic to decide whether a binary number is even or odd, is to check the least significant bit of the number. If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag, and increment even or odd number counter.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 2357H, 0A579H, 0C322H, 0C91EH, 0C000H, 0957H
COUNT EQU 006H
DATA ENDS
CODE SEGMENT
START:          XOR BX, BX
                XOR DX, DX
                MOV AX, DATA
                MOV DS, AX
                MOV CL, COUNT
                MOV SI, OFFSET LIST
AGAIN:          MOV AX, [SI]
                ROR AX, 01
                JC ODD
                INC BX
                JMP NEXT
ODD:            INC DX
NEXT:           ADD SI, 02
                DEC CL
                JNZ AGAIN
                MOV AH, 4CH
                INT 21H
                CODE ENDS
                END START
```

Program 3.5 *Listings*

Program 3.6

Write a program to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Solution Take the *i*th number in any of the registers. Rotate it left through carry. The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. If CF is 1, the number is negative; otherwise, it is positive.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 2579H, 0A500H, 0C009H, 0159H, 0B900H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:      XOR BX, BX
            XOR DX, DX
            MOV AX, DATA
            MOV DS, AX
            MOV CL, COUNT
            MOV SI, OFFSET LIST
AGAIN:      MOV AX, [SI]
            SHL AX, 01
            JC NEG
            INC BX
            JMP NEXT
NEG:        INC DX
NEXT:       ADD SI, 02
            DEC CL
            JNZ AGAIN
            MOV AH, 4CH
            INT 21H
            CODE ENDS
            END START
```

Program 3.6 *Listings*

The logic of Program 3.6 is similar to that of Program 3.5, hence comments are not given in Program 3.6 except for a few important ones.

Program 3.7

Write a program to move a string of data words from offset 2000H to offset 3000H the length of the string is 0FH.

Solution To write this program, we will use an important facility, available in the 8086 instruction set, i.e. move string byte/word instruction. We will also study the flexibility imparted by this instructions to the 8086 assembly language program. Let us first write the Program 3.7 for 8085, assuming that the string is available at location 2000H and is to be moved at 3000H.

```
                LXI H , 2000H
                LXI D , 3000H
                MVI C , 0FH
AGAIN :               MOV A , M
```

```
                  STAX   D
                  INX    H
                  INX    D
                  DCR    C
                  JNZ    AGAIN
                  HLT
```

An 8085 Program for Program 3.7

Now assuming DS is suitably set, let us write the sequence for 8086. At first using the index registers, the program can be written as given:

```
                  MOV    SI , 2000H
                  MOV    DI , 3000H
                  MOV    CX , 0FH
      AGAIN :     MOV    AX , [SI]
                  MOV    [DI], AX
                  ADD    SI, 02H
                  ADD    DI, 02H
                  DEC    CX
                  JNZ    AGAIN
                  HLT
```

An 8086 Program for Program 3.7

Comparing the above listings for 8085 and 8086, we may infer that every instruction in 8085 listing is replaced by an equivalent instruction of 8086. The above 8086 listing is absolutely correct but it is not efficient. Let us try to write the listings for the same purpose using the string instruction. Due to the assembler directives and the syntax, one may feel that the program is lengthy, though it eliminates four instructions for a MOVSW instruction.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV ES, AX
            MOV SI, SOURCESTRT
            MOV DI, DESTSTRT
            MOV CX, COUNT
            CLD
REP         MOVSW
            MOV AH, 4CH
            INT 21H
CODE ENDS
END START
```

Program 3.7 *An 8086 Program listing for Program 3.7 using String instruction*

Compare the above two 8086 listings. Both contain ten instructions. However, in case of the second program, the instruction for the initialisation of segment register and DOS interrupt are additional while the first one neither contains initialisation of any segment registers nor does it contain the DOS interrupt instruction. We can say that the first program uses 9 instructions, while the second one uses only 5 for implementing the same algorithm.

This program and the related discussions are aimed at explaining the importance of the string instructions and the method to use them.

Program 3.8

Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

Solution There exist a large number of sorting algorithms. The algorithm used here is called *bubble sorting*. The method of sorting is explained as follows. To start with, the first number of the series is compared with the second one. If the first number is greater than second, exchange their positions in the series otherwise leave the positions unchanged. Then, compare the second number in the recent form of the series with third and repeat the exchange part that you have carried out for the first and the second number, and for all the remaining numbers of the series. Repeat this procedure for the complete series (*n*−1) times. After (*n*−1) iterations, you will get the largest number at the end of the series, where n is the length of the series. Again start from the first address of the series. Repeat the same procedure right from the first element to the last element. After (*n*−2) iterations you will get the second highest number at the last but one place in the series. Continue this till the complete series is arranged in ascending order. Let the series be as given:

```
53 , 25 , 19, 02              n = 4
25 , 53 , 19, 02              1st operation
25 , 19 , 53 , 02             2nd operation
25 , 19 , 02 , 53             3rd operation
largest no.          ⇒        4 − 1 = 3 operations
19 , 25 , 02 , 53             1st operation
19 , 02 , 25 , 53             2nd operation
2nd largest number   ⇒        4 − 2 = 2 operations
02 , 19 , 25 , 53             1st operation
3rd largest number   ⇒        4 − 3 = 1 operations
```

Instead of taking a variable count for the external loop in the program like (*n* − 1), (*n* − 2), (*n*−3), ..., etc. It is better to take the count (*n* − 1) all the time for simplicity. The resulting program is given as shown.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 53H, 25H, 19H, 02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:       MOV AX, DATA
             MOV DS, AX
             MOV DX, COUNT-1
AGAIN0:      MOV CX, DX
             MOV SI, OFFSET LIST
AGAIN1:      MOV AX, [SI]
```

```
                CMP AX, [SI+2]
                JL  PR1
                XCHG [SI+2], AX
                XCHG [SI], AX
     PR1:       ADD SI, 02
                LOOP AGAIN1
                DEC DX
                JNZ AGAIN0
                MOV AH, 4CH
                INT 21H
     CODE       ENDS
                END START
```

Program 3.8 *Listings*

With a similar approach, the reader may write a program to arrange the string in descending order. For this, instead of the JL instruction in the above program, one will have to use a JG instruction.

Program 3.9

Write a program to perform a one byte BCD addition.

Solution It is assumed that the operands are in BCD form, but the CPU considers it hexadecimal and accordingly performs addition. Consider the following example for addition. Carry is set to be zero.

```
       9 2
     + 5 9
     -----
       E B      Actual result after addition considering hex.
                operands

       1 0 1 1
     + 0 1 1 0    As 0BH (LSD of addition) > 09, add 06 to it.
     ---------
     1 0 0 0 1    Least significant nibble of result (neglect the
                  auxiliary carry) → AF is set to 1
  0110 is added to most significant nibble of the result if it is greater
than 9 or AF is set.
                      1     Carry from previous digit (AF)
       E      →   1 1 1 0
              +   0 1 1 0
                  -------
 CF is set to 1   0 1 0 1   next significant nibble of result

 Result CF    Most significantLeast significant digit
    1               5                 1

     ASSUME CS:CODE, DS:DATA
     DATA SEGMENT
                OPR1 EQU 92H
                OPR2 EQU 52H
     RESULT DB 02 DUP(00)
     DATA ENDS
```

```
CODE SEGMENT
START:       MOV AX, DATA
             MOV DS, AX
             MOV BL, OPR1
             XOR AL, AL
             MOV AL, OPR2
             ADD AL, BL
             DAA
             MOV RESULT, AL
             JNC MSBO
             INC [RESULT+1]
MSBO:        MOV AH, 4CH
             INT 21H
CODE ENDS
END START
```

Program 3.9 *Listings*

In this program, the instruction DAA is used after ADD. Similarly, DAS can be used after SUB instruction. The reader may try to write a program for BCD subtraction for practice.

Program 3.10

Write a program that performs addition, subtraction, multiplication and division of the given operands. Perform BCD operation for addition and subtraction.

Solution Here we have directly given the routine for Program 3.10.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
             OPR1 EQU 98H
             OPR2 EQU 49H
             SUM DW 01 DUP(00)
             SUBT DW 01 DUP(00)
             PROD DW 01 DUP(00)
             DIVS DW 01 DUP(00)
DATA         ENDS
CODE         SEGMENT
START:       MOV AX, DATA
             MOV DS, AX
             MOV BL, OPR2
             XOR AL, AL
             MOV AL, OPR1
             ADD AL, BL
             DAA
             MOV BYTE PTR SUM, AL
             JNC MSBO
             INC [SUM+1]
MSBO:        XOR AL, AL
             MOV AL, OPR1
             SUB AL, BL
             DAS
```

```
                MOV BYTE PTR SUBT, AL
                JNB MSB1
                INC [SUBT+1]
MSB1:           XOR AL, AL
                MOV AL, OPR1
                MUL BL
                MOV WORD PTR PROD, AX
                XOR AH, AH
                MOV AL, OPR1
                DIV BL
                MOV WORD PTR DIVS, AX
                MOV AH, 4CH
                INT 21H
CODE            ENDS
                END START
```

Program 3.10 *Listings*

Program 3.11

Write a program to find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Solution The given string is scanned for the given byte. If it is found in the string, the zero flag is set; else, it is reset. Use of the SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found out. Note that, in this program, the code segment is written before the data segment.

```
ASSUME CS:CODE, DS:DATA
CODE SEGMENT
START:          MOV AX, DATA
                MOV DS, AX
                MOV ES, AX
                MOV CX, COUNT
                MOV DI, OFFSET STRING
                MOV BL, 00H
                MOV AL, BYTE1
SCAN1:          NOP
                SCASB
                JZ XXX
                INC BL
                LOOP SCAN1
XXX:            MOV AH, 4CH
                INT 21H
                CODE ENDS
DATA SEGMENT
BYTE1 EQU 25H
COUNT EQU 06H
STRING DB 12H, 13H, 20H, 20H, 25H, 21H
DATA ENDS
                END START
```

Program 3.11 *Listings*

Program 3.12

Write a program to convert the BCD numbers 0 to 9 to their equivalent seven segment codes using the look-up table technique. Assume the codes [7-seg] are stored sequentially in CODELIST at the relative addresses from 0 to 9. The BCD number (CHAR) is taken in AL.

Solution Refer to the explanation of the XLAT instruction. The statement of the program itself gives the explanation about the logic of the program.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
             CODELIST DB 34, 45, 56, 45, 23, 12, 19, 24, 21, 00
             CHAR EQU 05
             CODEC    DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
START:       MOV AX, DATA
             MOV DS, AX
             MOV BX, OFFSET CODELIST
             MOV AL, CHAR
             XLAT
             MOV BYTE PTR CODEC,AL
             MOV AH, 4CH
             INT 21H
CODE         ENDS
             END START
```

Program 3.12 *Listings*

Program 3.13

Decide whether the parity of a given number is even or odd. If parity is even set DL to 00; else, set DL to 01. The given number may be a multibyte number.

Solution The simplest algorithm to check the parity of a multibyte number is to go on adding the parity byte by byte with 00H. The result of the addition reflects the parity of that byte of the multibyte number. Adding the parities of all the bytes of the number, one will obtain the over all parity of the number:

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
             NUM DD 335A379BH
             BYTE_COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:       MOV AX, DATA
             MOV DS, AX
             MOV DH, BYTE_COUNT
             XOR AL, AL
             MOV CL, 00
             MOV SI, OFFSET NUM
NEXT_BYTE:   ADD AL, [SI]
             JP EVENP
```

```
                INC CL
EVENP:          INC SI
                MOV AL, 00
                DEC DH
                JNZ NEXT_BYTE
                MOV DL, 00
                RCR CL, 1
                JNC CLEAR
                INC DL
CLEAR:          MOV AH, 4CH
                INT 21H
CODE            ENDS
                END START
```

Program 3.13 *Listings*

The contents of CL are incremented depending upon either the parity for that byte is even or odd. If LSB of CL is 1, after testing for all bytes, it means the parity of the multibyte number is odd otherwise it is even and DL is modified correspondingly.

Program 3.14

Write a program for the addition of two 3 × 3 matrices. The matrices are stored in the form of lists (row wise). Store the result of addition in the third list.

Solution In the addition of two matrices, the corresponding elements are added to form the corresponding elements of the result matrix as shown:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11}+a_{12}+b_{12}+a_{13}+b_{13} \\ a_{21}+b_{21}+a_{22}+b_{22}+a_{23}+b_{23} \\ a_{31}+b_{31}+a_{32}+b_{32}+a_{33}+b_{33} \end{bmatrix}$$

$$[A] \quad + \quad [B] \quad = \quad [A + B]$$

The matrix *A* is stored in the memory at an offset MAT1, as given:
a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}, etc.

A total of 3 × 3 = 9 additions are to be done. The assembly language program is written as shown:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
                DIM EQU 09H
                MAT1 DB 01, 02, 03, 04, 05, 06, 07, 08, 09
                MAT2 DB 01, 02, 03, 04, 05, 06, 07, 08, 09
                RMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:          MOV AX, DATA
                MOV DS, AX
                MOV CX, DIM
                MOV SI, OFFSET MAT1
                MOV DI, OFFSET MAT2
                MOV BX, OFFSET RMAT3
```

```
NEXT:       XOR AX, AX
            MOV AL, [SI]
            ADD AL, [DI]
            MOV WORD PTR [BX], AX
            INC SI
            INC DI
            ADD BX, 02
            LOOP NEXT
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Program 3.14 *Listings*

Program 3.15

Write a program to find out the product of two matrices. Store the result in the third matrix. The matrices are specified as in the Program 3.14.

Solution The multiplication of matrices is carried out as shown:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}\,b_{11} + a_{12}\,b_{21} + a_{13}\,b_{31} & a_{12}\,b_{12} + a_{22}\,b_{12} + a_{13}\,b_{32} & a_{13}\,b_{13} + a_{12}\,b_{23} + a_{13}\,b_{23} \\ a_{21}\,b_{11} + a_{22}\,b_{21} + a_{23}\,b_{31} & a_{21}\,b_{12} + a_{22}\,b_{22} + a_{23}\,b_{32} & a_{21}\,b_{13} + a_{22}\,b_{23} + a_{23}\,b_{23} \\ a_{31}\,b_{11} + a_{32}\,b_{21} + a_{33}\,b_{31} & a_{31}\,b_{12} + a_{32}\,b_{22} + a_{33}\,b_{32} & a_{31}\,b_{13} + a_{32}\,b_{23} + a_{33}\,b_{23} \end{bmatrix}$$

The listings to carry out the above operation is given as shown:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
ROCOL EQU 03H
MAT1 DB 05H, 09H, 0AH, 03H, 02H, 07H, 03H, 00H, 09H
MAT2 DB 09H, 07H, 02H, 01H, 0H, 0DH, 7H, 06H, 02H
PMAT3 DW 09H DUP(?)
DATA ENDS
CODE SEGMENT
START:      MOV AX, DATA
            MOV DS, AX
            MOV CH, ROCOL
            MOV BX, OFFSET PMAT3
            MOV SI, OFFSET MAT1
NEXTROW:    MOV DI, OFFSET MAT2
            MOV CL, ROCOL
NEXTCOL:    MOV DL, ROCOL
            MOV BP, 0000H
            MOV AX, 0000H
            SAHF
NEXT_ELE:   MOV AL, [SI]
```

```
                MUL BYTE PTR[DI]
                ADD BP, AX
                INC SI
                ADD DI, 03
                DEC DL
                JNZ NEXT_ELE
                SUB DI, 08
                SUB SI, 03
                MOV [BX], BP
                ADD BX, 02
                DEC CL
                JNZ NEXTCOL
                ADD SI, 03
                DEC CH
                JNZ NEXTROW
                MOV AH, 4CH
                INT 21H
CODE ENDS
END START
```

Program 3.15 *Listings*

Program 3.16

Write a program to add two multibyte numbers and store the result as a third number. The numbers are stored in the form of the byte lists stored with the lowest byte first.

Solution This program is similar to the program written for the addition of two matrices except for the addition instruction.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BYTES EQU 08H
NUM1 DB 05, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H
NUM2 DB 04, 56H, 04H, 57H, 32H, 12H, 19H, 13H
NUM3 DB 0AH DUP(00)
DATA ENDS
CODE SEGMENT
START:          MOV AX, DATA
                MOV DS, AX
                MOV CX, BYTES
                MOV SI, OFFSET NUM1
                MOV DI, OFFSET NUM2
                MOV BX, OFFSET NUM3
                XOR AX, AX
NEXTBYTE:       MOV AL, [SI]
                ADC AL, [DI]
                MOV BYTE PTR[BX], AL
                INC SI
                INC DI
                INC BX
                DEC CX
                JNZ NEXTBYTE
                JNC NCARRY
                MOV BYTE PTR[BX], 01
NCARRY:         MOV AH, 4CH
                INT 21H
CODE ENDS
                END START
```

Program 3.16 *Listings*

Program 3.17

Write a program to add more than two multibyte numbers. The numbers are specified in a single list byte-wise one after another.

Solution In this program, all the numbers are stored in a single list byte-wise. The least significant byte of the first number is stored first, then the next significant byte and so on. After the most significant byte of the first number, the least significant byte of the second number will be stored. The series thus will end with the most significant byte of the last number. Let each number be of 8 bytes and 10 numbers are to be added. The list will contain $8 \times 10 = 80$ bytes. The result may have more than 8 bytes. Let us assume that the result requires 9 bytes to be stored. A separate string of 9 bytes is reserved for the result. The result is also stored in the same form as the numbers. The assembly language program for this problem is given as shown.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
BYTES EQU 04
NUMBERS EQU 02
NUMBERLIST DB 55H, 22H, 0BCH, 0FFH, 76H, 56H, 0FFH, 0F0H
RESULT DB BYTES+1 DUP(?)
DATA ENDS
CODE SEGMENT
START:          MOV AX, DATA
                MOV DS, AX
                XOR AX, AX
                MOV BL, BYTES
                MOV SI, OFFSET NUMBERLIST
                MOV DI, OFFSET RESULT
                MOV CL, BYTES
NEXTBYTE:       MOV CH, NUMBERS
NEXTNUM:        MOV AL, [SI]
                ADD SI, BYTES
                ADD [DI], AL
                JNC NOCARY
                INC BYTE PTR[DI+1]
NOCARY:         DEC CH
                JNZ NEXTNUM
                SUB SI, BYTES
                SUB SI, BYTES
                INC DI
                INC SI
```

```
                    DEC BL
                    JNZ NEXTBYTE
                    MOV AH,4CH
                    INT 21H
        CODE ENDS
                    END START
```

Program 3.17 *Listings*

Program 3.18

Write a program to convert a 16 bit binary number into equivalent BCD number.

Solution The program to convert the binary number into equivalent BCD number is developed below :

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BIN EQU
RESULT DW (?)
DATA ENDS
CODE SEGMENT
START :    MOV AX, DATA   ;  INITIALIZE DATA SEGMENT
           MOV DS, AX
           MOV BX, BIN
           MOV AX, 0      ;  INITIALIZE TO 0
           MOV CX, 0      ;  INITIALIZE TO 0
CONTINUE : CMP BX, 0      ;  COMPARISION FOR ZERO BINARY NUMBER.
           JZ ENDPROG     ;  IF ZERO END THE PROGRAM
           DEC BX         ;   DECREMENT BX BY 1
           MOV AL, CL
           ADD AL, 1      ;   ADD 1 TO AL
           DAA            ;  DECIMAL ADJUST AFTER ADDITION
           MOV CL, AL     ;  STORING RESULT IN CL REGISTER
           MOV AL, CH
           ADC AL, 00H    ;  ADD WITH CARRY
           DAA
           MOV CH, AL     ;  STORING RESULT IN CH REGISTER
           JMP CONTINUE
ENDPROG :  MOV RESULT, CX ;  STORING RESULT IN DATA SEGMENT
           MOV AH, 4CH
           INT 21H
           CODE ENDS
           END START
```

Program 3.18 *Listings*

Program 3.19

Write a program to convert a BCD number into an equivalent binary number.

Solution A program to convert a BCD number into binary equivalent number is developed below.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
BCD_NUM EQU 4576H
BIN_NUM DW (?)
DATA ENDS
CODE SEGMENT
START :       MOV AX, DATA    ; INITIALIZE DATA SEGMENT
              MOV DS, AX
              MOV BX, BCD_NUM ; BX IS NOW HAVING BCD NUMBER
              MOV CX, 0       ; INITIALIZATION
CONTINUE :    CMP BX, 0       ; COMPARISON TO CHECK BCD NUM IS ZERO
              J2 ENDPROG      ; IF ZERO END THE PROGRAM
              MOV AL, BL      ; 8 LSBs OF NUMBER IS TRANSFERRED TO
                                AL
              SUM AL, 1       ; SUBTRACT ONE FROM AL
              DAS             ; DECIMAL ADJUST AFTER SUBTRACTION
              MOV BL, AL      ; RESULT IS STORED IN BL
              MOV AL, BH      ; 8 MSB IS TRANSFERRED TO AL
              SBB AL, 00H     ; SUBTRACTION WITH BORROW
              DAS             ; DECIMAL ADJUST AFTER SUBTRACTION
              MOV BH, AL      ; RESULT BACK IN BH REGISTER
              INC CX          ; INCREMENT CX BY 1
              JMP CONTINUE
ENDPROG :     MOV BIN_NUM, CX ; RESULT IS STORED IN DATA SEGMENT
              MOV AH, 4CH     ; TERMINATION OF PROGRAM
              INT 21H         ; TERMINATION OF PROGRAM
              CODE ENDS
              END START
```

Program 3.19 *Listings*

Program 3.20

Write a program to convert an 8 bit binary number into equivalent gray code.

Solution A program to convert an 8 bit binary number into equivalent gray code is written below.

Binary $\rightarrow$ B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0

Gray $\rightarrow$ G_7 G_6 G_5 G_4 G_3 G_2 G_1 G_0

$G_7 = B_7$

$G_i = B_i \oplus B_{i+1}$

Where $i = 0$ to 6

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
NUM EQU 34H
RESULT DB (?)
```

```
DATA ENDS
START :      MOV AX, DATA     ;  INITIALIZE DATA SEGMENT
             MOV DS, AX
             MOV AL, NUM      ;  NUMBER IS TRANSFERRED TO AL
             MOV BL, AL
             CLC              ;  CLEAR CARRY FLAG
             RCR AL, 1        ;  ROTATE THROUGH CARRY THE CONTENT
                                     OF AL
             XOR BL, .AL      ;  XORING BL AND AL TO GET GRAY CODE
             MOV RESULT, BL   ;  STORING RESULT IN DMS
             MOV AH,4CH
             INT 21H
             CODE ENDS
             END START
```

Program 3.20 *Listings*

Program 3.21

Find square root of a two digit number. Assume that the number is a perfect square.

Solution The 2 digit number of which the square root is to be found out is considered as a perfect square. The program for this problem is presented below:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
NUM EQU 36
RESULT DB (?)
DATA ENDS
CODE SEGEMENT
START :      MOV AX, DATA      ;  INITIALIZE DATA SEGMENT
             MOV DS, AX
             MOV CL, NUM       ;  NUMBER IS TRANSFERRED TO CL
             MOV BL, 1         ;  BL IS INITIALISE TO 1
             MOV AL, 0         ;  AL INITIALISE TO 0
UP :         CMP CL, 0         ;  CHECK FOR ZERO NUMBER
             JZ ZRESULT        ;  IF ZERO THEN GO TO ZRESULT LABEL
             SUB CL, BL        ;  IF NOT ZERO SUBSTRACT BL FROM CL
             INC AL            ;  INCREMENT THE CONTENT OF AL
             ADD BL, 02        ;  ADD TWO TO REGISTER BL
             JMP UP            ;  GO BACK TO LABEL UP
ZRESULT :    MOV RESULT, AL    ;  RESULT IS SAVED IN DATA SEGMENT
             MOV AH, 4CH
             INT 21H
             CODE ENDS
             END START
```

Program 3.21 *Listings*

3.4.2 Programs to Utilize the Resources of an IBM Microcomputer Using DOS Function Calls

In this section, we will study the method of utilizing the hardware resources of an IBM microcomputer system working under DOS, using assembly language. In a computer system, each peripheral is assigned an address. The data can be written to or read from the peripheral using the write or read instructions, e.g. MOV, IN, OUT, etc. along with the address of the particular peripheral. The disk operating system has a unique way of accessing the hardware resources through the interrupt INT 21H.

For example, suppose we want to display a message on the CRT, then we will have to prepare a string of the message, then execute the instruction INT 21H with the function value 09H in AH and DS : DX set as a pointer to the start of the string.

Appendix-B tabulates the different function values and their purposes under the INT 21 H interrupt. With the help of the information in Appendix-B and the instruction set of 8086, we are able to access the hardware resources of the system like CRT, keyboard, hard disk, floppy disk, memory, etc. Also, the software resources like directory structure, file allocation table, can be referred by using the information in Appendix-B. With the help of a few simple programs, we now explain, how to use the particular resource. It is assumed that the computer system is working under DOS.

Program 3.22

Display the message "The study of microprocessors is interesting." on the CRT screen of a microcomputer.

Solution A program to display the string is given as follows:

```
ASSUME     CS :CODE, DS :DATA
DATA       SEGMENT
MESSAGE    DB ODH, OAH,"  STUDY OF MICROPROCESSORS IS INTERESTING",
                          ODH, OAH, "$"
                          ;PREPARING STRING OF THE MESSAGE
DATA       ENDS
CODE       SEGMENT
START:     MOV AX, DATA   ;INITIALIZE DS
           MOV DS, AX
           MOV AH, 09H    ;SET FUNCTION VALUE  FOR DISPLAY
           MOV DX, OFFSET MESSAGE
           INT 21H        ;POINT TO MESSAGE AND RUN
           MOV AH, 4CH    ;THE INTERRUPT
           INT 21H        ;RETURN TO DOS
CODE       ENDS           ;STOP
           END START
```

Program 3.22 *Listings*

The above listing starts with the usual statement ASSUME. In the data segment, the message is written in the form of a string of the message characters and cursor control characters. The characters 0AH and 0DH are the line feed and carriage feed characters. The "$" is the string termination character. At the end of every message to be displayed the "$" must be there. Otherwise, the computer loses the control, as it is unable to find the end of the string. The character 0DH brings the cursor to next line. The character 0AH brings cursor to the next position (column wise). In case of DB operator, the characters written in the statement in inverted double commas are built in the form of their respective ASCII codes in the allotted memory bytes for the string.

Then the data segment that contains the message string is initialised. The register AH is loaded with the function value 09H for displaying the message on the CRT screen. The instruction INT 21H after execution, causes the message to be displayed. The register DX points to the message in the data segment. After the message is displayed the control is returned to DOS prompt.

Program 3.23

Write a program to open a new file KMB.DAT in the current directory and drive. If it is successfully opened, write 200H bytes of data into it from a data block named BLOCK. Display a message, if the file is not opened successfully.

Solution This type of programs, written for the utilization of resources of a computer system does not require much of logic. These programs contain the specific function calls along with some instructions to load the registers to prepare the environment (required data) for the interrupt call. A flow chart for Program 3.23 is shown in Fig. 3.14. It is up to the application designer to combine these programs with the actual application programs.

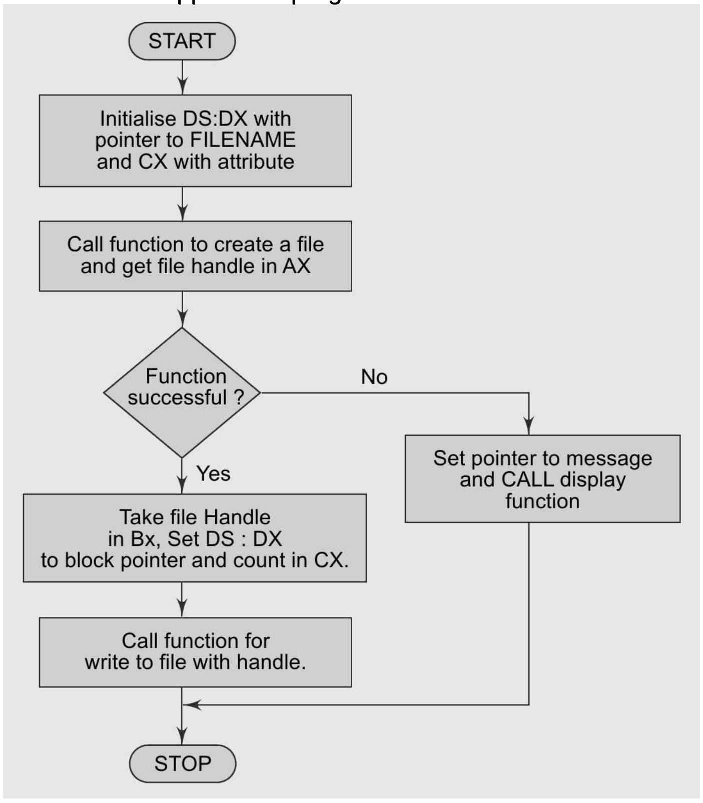
START
Initialise DS:DX with pointer to FILENAME and CX with attribute
Call function to create a file and get file handle in AX
Function successful ?
No
Yes
Set pointer to message and CALL display function
Take file Handle in Bx, Set DS : DX to block pointer and count in CX.
Call function for write to file with handle.
STOP

Fig. 3.14 *Flow Chart for Program 3.23*

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
        DATABLOCK DB 200H DUP (?)
        FILENAME DB "KMB.DAT", "$"
        MESSAGE DB 0AH, 0DH, "FILE NOT CREATED
        SUCCESSFULLY", 0AH, 0DH, "$"
```

```
DATA    ENDS
CODE    SEGMENT
START:  MOV AX, DATA               ;INTIALIZE DS
        MOV DS, AX
        MOV DX, OFFSET FILENAME    ;OFFSET OF FILE NAME
        MOV CX, 00H                ;FILE ATTRIBUTE IN CX,REFER TO
        MOV AH, 3CH                ;APPENDIX B FUNCTION 3CH
        INT 21H                    ;T0 CREATE A FILE, FOR more DE-
                                   ;TAILS.
        JNC WRITE                  ;IF FILE IS CREATED SUCCESSFULLY
                                   ;WRITE DATA IN TO IT
        MOV AX, DATA
        MOV DS, AX
        MOV DX, OFFSET MESSAGE     ;ELSE DISPLAY THE
        MOV AH, 09H                ;MESSAGE AND RETURN TO
        INT 21H                    ;DOS PROMPT WITHOUT WRITING
        JMP STOP                   ;DATA IN FILE
WRITE:  MOV BX, AX                 ;FILE HANDLE IN BX
        MOV CX, 0200H              ;LENGTH OF THE DATA BYTES TO BE
                                    WRITTEN
        MOV DX, OFFSET DATABLOCK   ;OFFSET OF THE SOURCE BLOCK
        MOV AH, 40H                ;IN DX AND USE 40H FUNCTION
        INT 21H
STOP:   MOV AH, 4CH
        INT 21H
CODE    ENDS
        END START
```

Program 3.23 *Listings*

Program 3.24

Write a program to load a file KMB.EXE in the memory at the CS value of 5000H with zero relocation factor. The file is just to be observed and not to be executed (overlay loading).

Solution This type of loading of a file is called as overlay loading. The function value 4BH in AH and 03 in AL serves the purpose. The program is given as shown. The reader may refer to Appendix-B for details of the function calls.

```
ASSUME          CS:CODE;DS:DATA
DATA            SEGMENT
LODPTR  DB 00, 50H, 00, 00
MESSAGE DB 0AH,0DH, "LOADING FAILURE", 0AH,0DH, "$"
FILENAME DB "PROG3-5.EXE","$"
DATA            ENDS
CODE            SEGMENT
START:          MOV AX, DATA
                MOV DS, AX              ;SET DS:DX TO STRING CONTAINING
                MOV DX, OFFSET FILENAME;FILE NAME
                MOV BX, OFFSET LODPTR   ;SET ES&BX TO POINT
```

```
               MOV AX, SEG LODPTR     ;A BLOCK CONTAINING
               MOV ES, AX             ;CS & RELOCATION FACTOR
               MOV AX, 4B03H          ;LOAD FUNCTION VALUE AND
               INT 21H
               JNC OKAY               ;IF LOADING IS NOT SUCCESSFULL,
               MOV AX, DATA
               MOV DS, AX
               MOV DX, OFFSET MESSAGE ;DISPLAY THE FAILURE
               MOV AH, 09H            ;MESSAGE.
               INT 21H
OKAY:          MOV AH, 4CH            ;RETURN TO DOS PROMPT.
               INT 21H
CODE           ENDS
               END START
```

Program 3.24 *Listings*

Refer to **MS DOS Encyclopedia by Ray Duncan** for more information on the function calls.

Program 3.25

Write a program using the AUTOEXEC.BAT file that hangs the computer and waits for the entry of the string 'ROY BHURCHANDI' from the key board and then returns to the DOS prompt, if the string is entered. The key board entries are not echoed (not displayed on the screen).

Solution The file AUTOEXEC.BAT is automatically run whenever the computer is reset. The listings for this program may be written in any assembly language file. This file should then be assembled, i.e.. EXE file should be prepared as already discussed in this chapter. The name of this .EXE file should be written in the AUTOEXEC.BAT file with the complete path of this .EXE file. The AUTOEXEC.BAT file must be available in the main (root) directory and default drive.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
STRING DB "ROY BHURCHANDI"
LENGTH DW OEH
BUFFER DB OFH DUP(?)
MESSAGE DB OAH, ODH, "SORRY!", OAH, ODH, "$"
DATA ENDS
CODE SEGMENT
START:     MOV AX, DATA           ; Initialize DS
           MOV DS, AX
WAIT:      MOV CX, LENGTH         ; String length in CX
           MOV DI, OFFSET BUFFER  ; DI points to buffer for keyboard
NXTCHAR:   MOV AH, 08H            ; entries under function 21H
           INT 21H
           CMP AL, ODH            ; If entries are over proceed for
           JE STOP                ; string comparison else store the
                                    character
           MOV [DI], AL           ; in the buffer and
           INC DI                 ; increment DI,
           DEC CX                 ; decrement CX for next entry
           JNZ NXTCHAR            ; Go for entering next character
```

```
STOP:     MOV CX, LENGTH          ; Set CX to length again.
          MOV SI, OFFSET STRING   ; Set SI and DI for string
          MOV DI, OFFSET BUFFER   ; comparison.
          MOV AX, SEG BUFFER      ; Set ES as segment pointer
          MOV ES, AX              ; to the string.
REP       CMPSB                   ; Compare the string with the buffer
          JNZ SORRY               ; If string does not match with buffer
                                  ; display 'Sorry!', otherwise return
          MOV AH, 4CH             ; to DOS prompt.
          INT 21H
SORRY:    MOV DX, OFFSET MESSAGE  ; Set offset pointer to display
                                    'Sorry!'.
          MOV AH, 09H             ; Set function value to 09H under
          INT 21H                 ; DOS interrupt 21h.
          JMP WAIT                ; Wait for the next entries.
CODE      ENDS
          END START
```

Program 3.25 *Listings*

In these programs, setting the supporting parameters and the function values is of prime importance. One may go through Appendix-B and try to write more and more programs for different functions available under INT 21H of DOS.

SUMMARY

This chapter starts with simple programs written for hand-coding. Further, hand-coding procedures of a few example instructions are studied. Advantages of the assembly language over machine language are then presented in brief. With an overview of the assembler operation, we have initiated the discussion of assembly language programming. The procedures of writing, entering and coding the assembly language programs are then discussed in brief. Further, DOS debugging program - DEBUG, a basic tool for trouble-shooting the assembly language programs, is discussed briefly with an emphasis on the most useful commands. Then we have presented various examples of 8086/8088 programs those highlight the actual use and the syntax of the instructions and directives. Finally, a few programs that enable the programmer to access the computer system resources using DOS function calls are discussed. We do not claim that each program presented here is the most efficient one, rather it just suggests a way to implement the algorithm asked in the problem. There may be a number of alternate algorithms and program listings to implement a logic but amongst them a programmer should choose one which requires minimum storage, execution time and complexity.

EXERCISES

3.1 Find out the machine code for following instructions.

(i) ADC AX,BX	(ii) OR AX,[0500H]	(iii) AND CX,[SI]
(iv) TEST AX,5555H	(v) MUL [SI+5]	(vi) NEG 50[BP]
(vii) OUT DX,AX	(viii) LES DI,[0700H]	(ix) LEA SI,[BX+500H]

(x) SHL [BX+2000],CL	(xi) RET 0200H	(xii) CALL 7000H
(xiii) JMP 3000H:2000H	(xiv) CALL [5000H]	(xv) DIV [5000H]

3.2 Describe the procedure for coding the intersegment and intrasegment jump and call instructions.
3.3 Enlist the advantages of assembly language programming over machine language.
3.4 What is an assembler?
3.5 What is a linker?
3.6 Explain various DEBUG commands for troubleshooting executable programs. (Refer to MSDOS Encyclopedia by Ray Duncan.)
3.7 What are the DOS fuction calls?
3.8 Write an ALP to convert a four digit hexadecimal number to decimal number.
3.9 Write an ALP to convert a four digit octal number to decimal number.
3.10 Write an ALP to find out ASCII codes of alphanumeric characters from a look up table.
3.11 Write an ALP to change an already available ascending order byte string to descending order.
3.12 Write an ALP to perform a 16-bit increment operation using 8-bit instructions.
3.13 Write an ALP to find out average of a given string of data bytes neglecting fractions.
3.14 Write an ALP to find out decimal addition of sixteen four digit decimal numbers.
3.15 Write an ALP to convert a four digit decimal number to its binary equivalent.
3.16 Write an ALP to convert a given sixteen bit binary number to its GRAY equivalent.
3.17 Write an ALP to find out transpose of a 3x3 matrix.
3.18 Write an ALP to find out cube of an 8-bit hexadecimal number.
3.19 Write an ALP to display message 'Happy Birthday!' on the screen after a key 'A' is pressed.
3.20 Write an ALP to open a file in the drive C of your hard disk , accept 256 byte entries each separated by comma and then store all these 256 bytes into the opened file. Issue suitable messages where-ever necessary.
3.21 Write an ALP to print a message 'The Printer Is Busy' on to a dot matrix printer.
3.22 Write an ALP to load a file from hard disk of your system into RAM system at segment address 5000H with zero relocation factor.
3.23 Write an ALP that goes on accepting the keyboard entries and displays them on line on the CRT display. The control escapes to DOS prompt if enter key is pressed.
3.24 Write an ALP to set interrupt vector of type 50H to an address of a routine ISR in segment CODE1.
3.25 Write an ALP to set and get the system time. Assume arbitary time for setting the system time.
3.26 What is the function 4CH under INT 21H?

Excercises 3.8 to 3.25 may be executed using a Windows based PC and any version of MASM as a part of Lab exercise.