```
INT 21H          ; Display digit
LOOP DISP
POP AX           ; Restore registers
POP BX
POP CX
POP DX
RET
ENDP
END
```

```
C:\tasm\tasm s_bta4d.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_bta4d.asm
Error messages:       None
Warning messages:     None
Passes:                  1
Remaining memory:     410k

C:\tasm\tlink s_bta4d.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_bta4d
10940
```

3.17.2 Routine to Convert ASCII to Binary

When we accept decimal number from keyboard we get ASCII code of each decimal digit. This information from the keyboard must be converted from ASCII to binary. When a single key is pressed conversion can be achieved by subtracting 30H. However, when more than one key is typed conversion from ASCII to binary requires 30H to be subtracted, but there is additional step. After subtracting 30H, the number is added to the result after the prior result is first multiplied by 10.

∴ 256 Decimal → 100H

Keystroke	Keyinput	SUB 30H	Calculations	
2 →	32H →	32H–30H →	02	
			× 0A	Multiply by 10
			14H	
			+	
5 →	35H →	35H–30H →	05H	Add next digit
			19H	
			× 0AH	Multiply by 10
			FAH	
			+	
6 →	36H →	36H–30H →	06H	Add next digit
			100H ←	Result

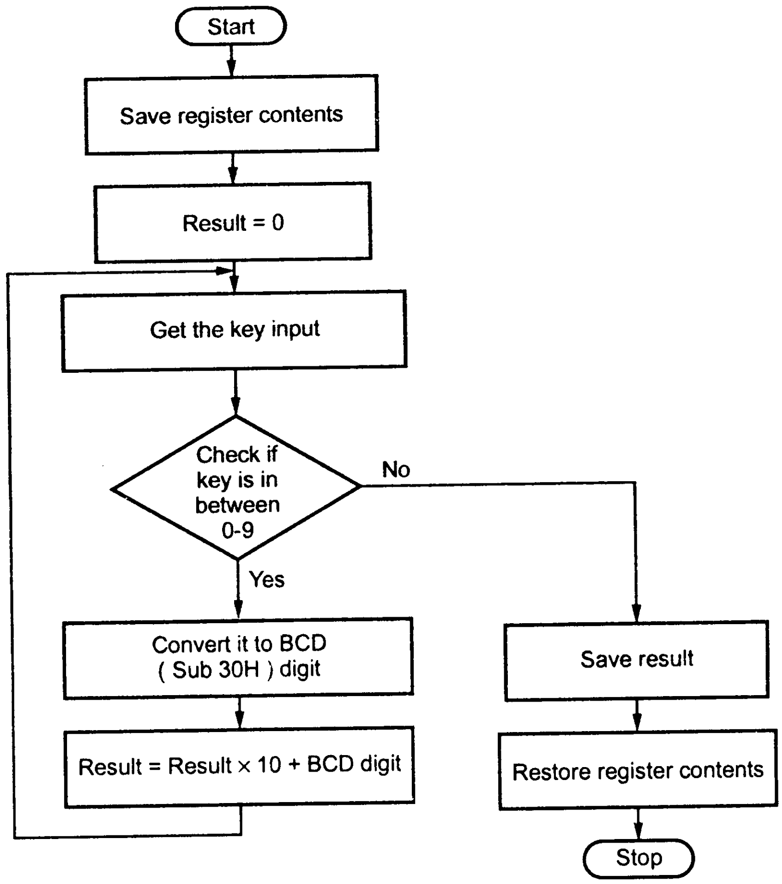
∴ 256 Decimal → 100H

Let us see the algorithm for converting number from ASCII to binary code.

Algorithm

1. Save contents of all registers which are used in the routine.
2. Make binary result = 0.
3. Subtract 30H from the character typed on the keyboard to convert it to BCD.
4. Multiply the result by 10, and then add the new BCD digit.
5. Repeat steps 2 and 3 until the character typed is not an ASCII coded number.
6. Restore register contents.

Flowchart



Routine : Convert BCD number from keyboard to its Hex equivalent.

```
; Routine to convert ASCII coded decimal from keyboard into its HEX
equivalent

ATB PROC NEAR
        PUSH CX             ; Save registers
        PUSH BX
        PUSH AX
        MOV CX, 10          ; Load 10 decimal in CX
        MOV BX, 0           ; Clear result
BACK:   MOV AH,01H          ; [Read key
        INT 21H             ; with echo]
        CMP AL,'0'
        JB SKIP             ; Jump if below '0'
        CMP AL,'9'
        JA SKIP             ; Jump if above '9'
        SUB AL, 30H         ; Convert to BCD
        PUSH AX             ; Save digit
        MOV AX, BX
        MUL CX              ; Multiply previous result by 10
        MOV BX, AX          ; Get the result in BX
        POP AX              ; Retrieve digit
        MOV AH, 00H
        ADD BX, AX          ; Add digit value to result
        JMP BACK            ; Repeat
SKIP:   MOV NUMBER,BX       ; Save the result in NUMBER
        POP AX              ; Restore registers
        POP BX
        POP CX
        RET
        ENDP
```

Sample Program

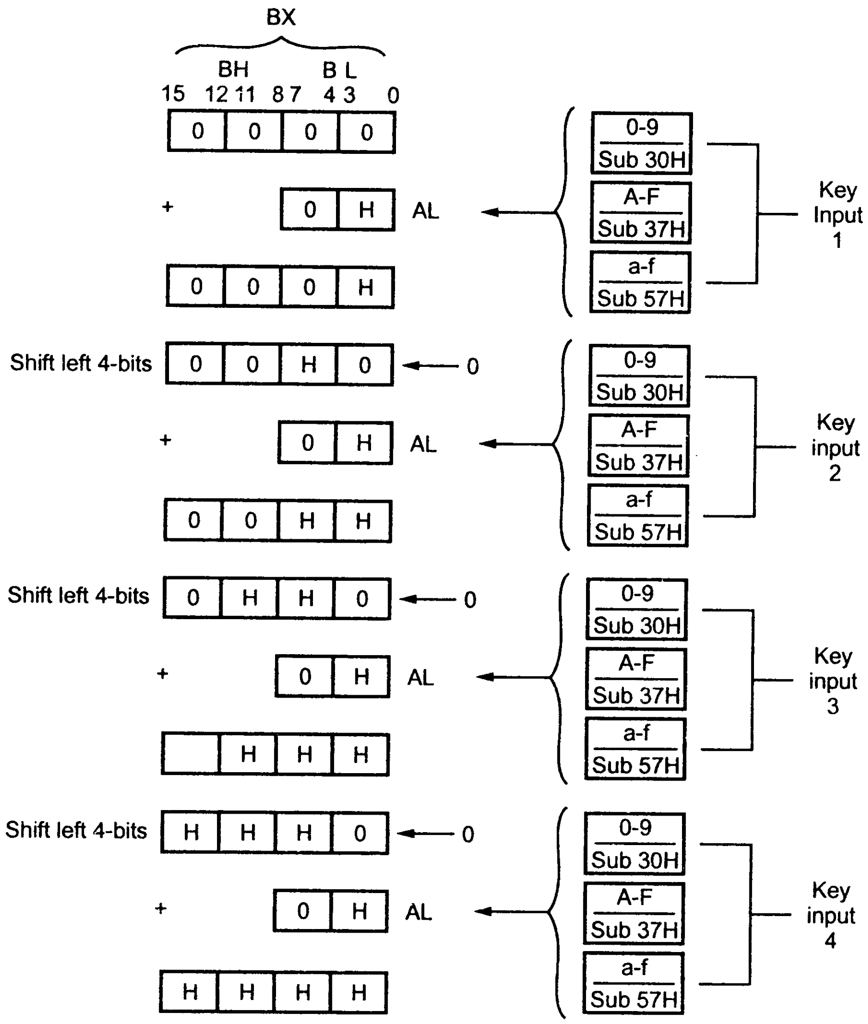
```
; Sample program to convert ASCII coded decimal from keyboard into
its HEX equivalent

.MODEL SMALL
.DATA
    NUMBER DW ?             ; Define number
.CODE
START:  MOV AX, @DATA       ; [Initialize
        MOV DS, AX          ; data segement]
        CALL ATB            ; convert ASCII coded decimal from
                            ; keyboard into its HEX equivalent

        MOV AH, 4CH         ; [Exit to
        INT 21H             ;   DOS]
ATB PROC NEAR
        PUSH CX             ; Save registers
        PUSH BX
        PUSH AX
        MOV CX, 10          ; Load 10 decimal in CX
        MOV BX, 0           ; Clear result
BACK:   MOV AH,01           ; [Read key
        INT 21H             ;  with echo]
        CMP AL,'0'
        JB SKIP             ; Jump if below '0'
        CMP AL,'9'
```

```
        JA SKIP          ; Jump if above '9'
        SUB AL, 30H      ; Convert to BCD
        PUSH AX          ; Save digit
        MOV AX, BX       ; Multiply previous result by 10
        MUL CX
        MOV BX, AX       ; Get the result in BX
        POP AX           ; Retrieve digit
        MOV AH, 00H
        ADD BX, AX       ; Add digit value to result
        JMP BACK         ; Repeat
SKIP:   MOV NUMBER,BX    ; Save the result in NUMBER

        POP AX           ; Restore registers
        POP BX
        POP CX
        RET
        ENDP
        END
```

```
C:\tasm\tasm s_atb.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:     s_atb.asm
Error messages:      None
Warning messages:    None
Passes:                 1
Remaining memory:    410k

C:\tasm\tlink s_atb.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_atb
1234
```

3.17.3 Routine to Read Hexadecimal Data

We know that hexadecimal numbers range from 0 to 9 and from A to F. The keyboard gives ASCII codes for these hexadecimal numbers. It gives 30H to 39H for numbers 0 to 9 and gives 41H to 46H for A to F letters or gives 61H to 66H for a to f letters. Hence, to convert ASCII input from keyboard to corresponding hexadecimal number we have to first check whether it is a number or letter and then if letter whether it is a small letter or capital letter and accordingly convert it into hexadecimal number.

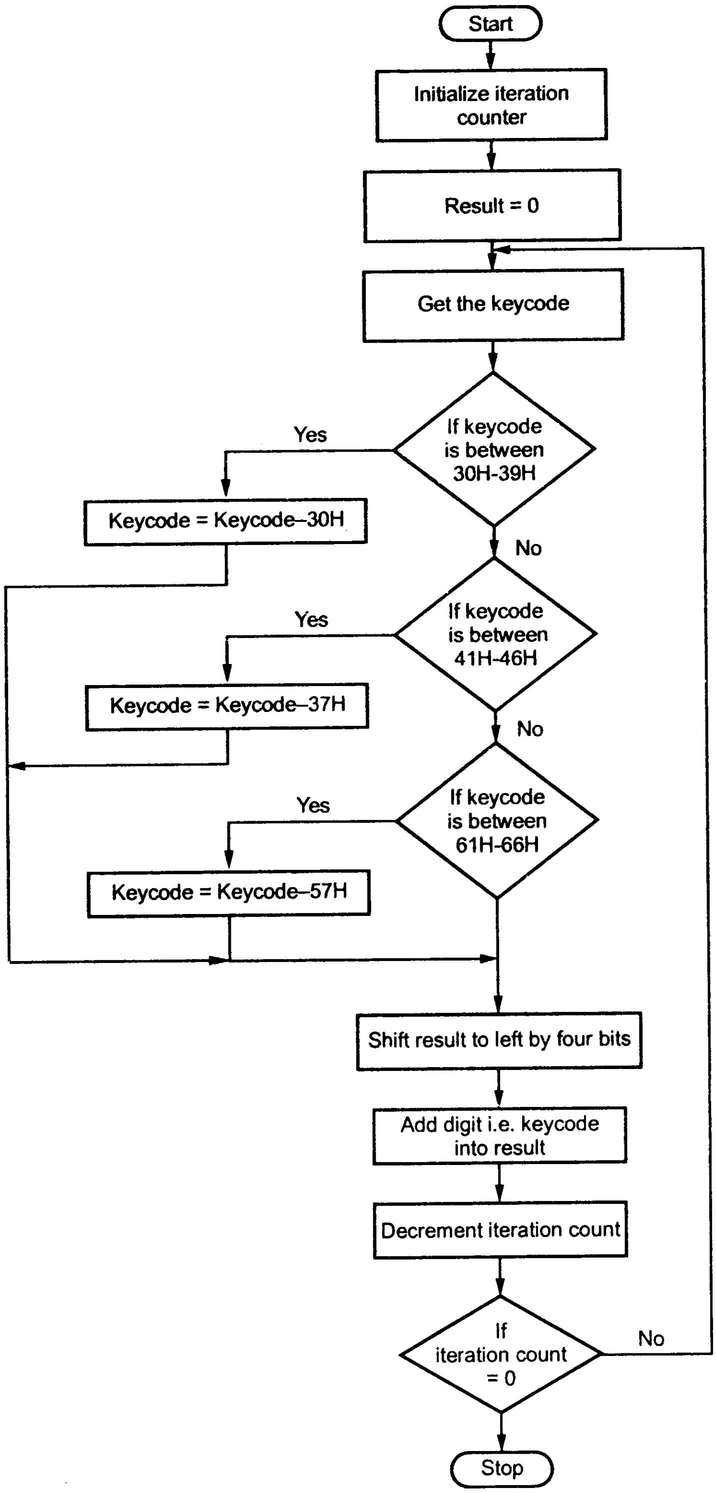


Note : H represents any hexadecimal digit (0-F).

Algorithm

1. Save registers
2. Make result = 0
3. Get the ASCII code of character from keyboard and
   - Subtract 30H from it if character is 0 - 9
   - Subtract 37H from it if character is A - F
   - Subtract 57H from it if character a - f
4. Shift the result by 4-bits and add digit to pack binary digits.
5. Repeat steps 2 and 3 four times to get 4-digit hex number.
6. Restore registers.

Flowchart


Start
Initialize iteration counter
Result = 0
Get the keycode
If keycode is between 30H-39H
Yes
Keycode = Keycode–30H
No
If keycode is between 41H-46H
Yes
Keycode = Keycode–37H
No
If keycode is between 61H-66H
Yes
Keycode = Keycode–57H
Shift result to left by four bits
Add digit i.e. keycode into result
Decrement iteration count
If iteration count = 0
No
Stop

Routine : Reading hexadecimal data

Returns : Hex number in variable number

```
; Routine to read 4-digit Hex number from the keyboard

R_HEX PROC NEAR

        PUSH CX                 ; Save registers
        PUSH BX
        PUSH AX
        PUSH SI
        MOV CL, 04              ;  Load shift count
        MOV SI, 04              ;  Load iteration count
        MOV BX, 0               ;  Clear result
BACK:   MOV AH, 01              ; [Read a key
        INT 21H                 ;  with echo]
        CALL CONV               ;  convert to binary
        SHL BX, CL              ; [pack four
        ADD BL, AL              ;  binary digits
        DEC SI                  ;  as 16-bit
        JNZ BAC                 ;  number]
        MOV NUMBER, BX          ; Save result at NUMBER
        POP SI                  ; Restore registers
        POP AX
        POP BX
        POP CX
        RET
        ENDP

; The procedure to convert contents of AL into hexadecimal
equivalent

CONV PROC NEAR
            CMP AL,'9'
            JBE SUBTRA30        ; If number is between 0 through 9
            CMP AL,'a'
            JB SUBTRA37         ; If letter is uppercase
            SUB AL, 57H         ; Subtract 57H if letter is lowercase
            JMP LAST1
SUBTRA30:   SUB AL, 30H         ; Convert number
            JMP LAST1
SUBTRA37:   SUB AL, 37H         ; Convert uppercase letter
LAST1:      RET
CONV        ENDP
```

Sample Program

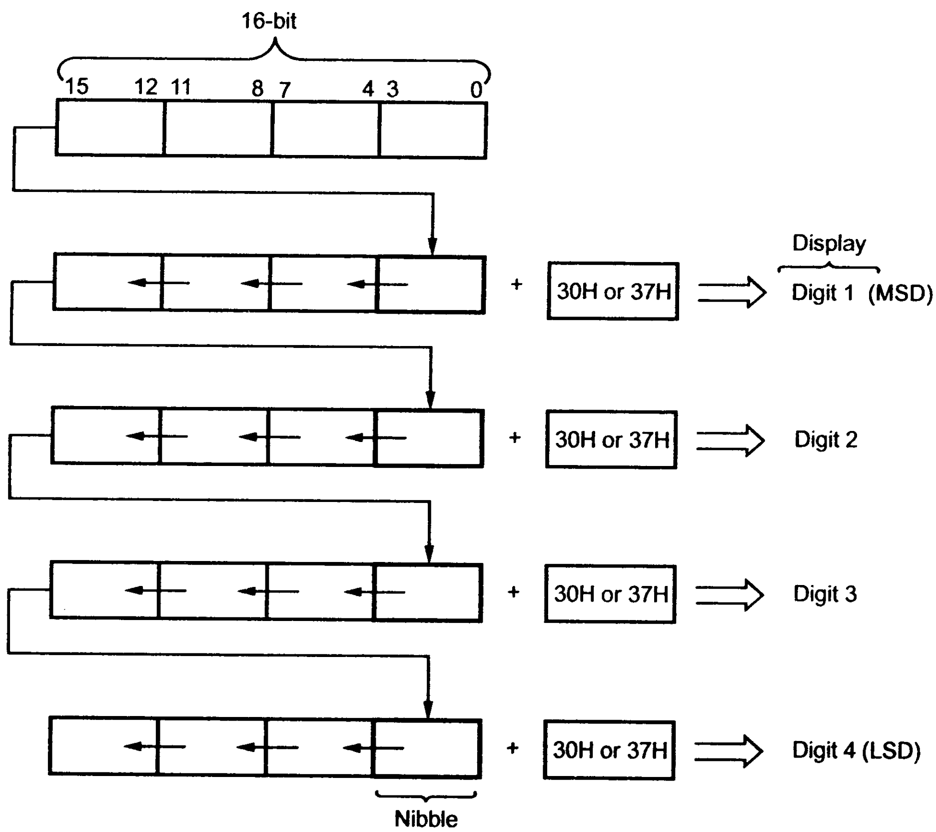
```
; Sample example to read 4-digit Hex number from the keyboard

.MODEL SMALL                    ; Select small model
.STACK 100                      ; Initialise stack
```

```
.DATA                            ; Start data segment
        NUMBER DW?               ; Define NUMBER
.CODE                            ; Start code segment
START:MOV AX, @DATA              ; [Initialize
        MOV DS, AX               ;  data segment]
        CALL R_HEX               ; Read 4-digit hex number
        MOV AH, 4CH              ; [Exit to
        INT 21H                  ;  DOS]
R_HEX PROC NEAR
        PUSH CX                  ; Save registers
        PUSH BX
        PUSH AX
        PUSH SI
        MOV CL, 04               ;  Load shift count
        MOV SI, 04               ;  Load iteration count
        MOV BX, 0                ;  Clear result
BAC:    MOV AH, 01               ; [Read a key
        INT 21H                  ;  with echo]
        CALL CONV                ;  Convert to binary
        SHL BX, CL               ; [Pack four
        ADD BL, AL               ; binary digits
        DEC SI                   ;  as 16-bit
        JNZ BAC                  ;  number]
        MOV NUMBER, BX           ; Save result at NUMBER
        POP SI                   ; Restore registers
        POP AX
        POP BX
        POP CX
        RET
        ENDP

; The procedure to convert contents of AL into hexadecimal
equivalent

CONV PROC NEAR
            CMP AL,'9'
            JBE SUBTRA30         ; If number is between 0 through 9
            CMP AL,'a'
            JB SUBTRA37          ; If letter is uppercase
            SUB AL, 57H          ; Subtract 57H if letter is
                                 ; Lowercase
            JMP LAST1
SUBTRA30:   SUB AL, 30H          ; Convert number
            JMP LAST1
SUBTRA37:   SUB AL, 37H          ; Convert uppercase letter
LAST1:      RET
CONV        ENDP
            END
```

```
C:\tasm\tasm s_rdhex.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_rdhex.asm
Error messages:       None
Warning messages:     None
Passes:                  1
Remaining memory:     410k

C:\tasm\tlink s_rdhex.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_rdhex
12AB
```

3.17.4 Routine to Display Hexadecimal Data

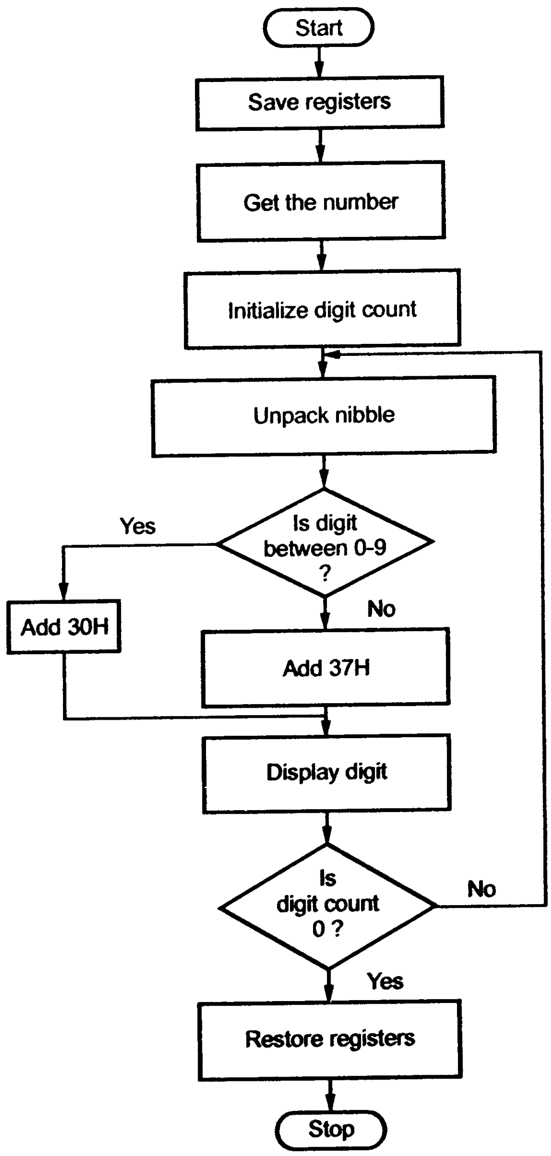
To display hexadecimal data we have to first unpack each digit (nibble) in the given number. Then by adding 30H to digit having number between 0 to 9 and by adding 37H to digit having letter between A to F we can get the ASCII equivalent of given hexadecimal number. This can be achieved by rotating number left (nibble by nibble) and adding 30H or 37H into it. By rotating left we can display left most digit (MSD) first.



Algorithm

1. Save registers.
2. Get the number and unpack digit from it.
3. Add 30H if digit is 0 - 9 or add 37H if digit is A - F to get the ASCII code of digit.
4. Display digit.
5. Repeat steps 2, 3 and 4.
6. Restore registers.

Flowchart



Routine

```
; Routine to display 4-digit hex number in AX

D_HEX PROC NEAR

     PUSH DX            ; Save registers
     PUSH CX
     PUSH AX
     MOV CL, 04H        ; Load rotate count
     MOV CH, 04H        ; Load digit count
BACK:    ROL AX, CL     ; Rotate digits
     PUSH AX            ; Save contents of AX
     AND AL, 0FH        ; [Convert
     CMP AL,9           ;  number
     JBE ADD30          ; to
     ADD AL, 37H        ; its
     JMP DISP           ; ASCII
ADD30:
     ADD AL,30H         ; equivalent]
DISP:    MOV AH,02H
     MOV DL,AL          ; [Display the
     INT 21H            ;   number]
     POP AX             ; Restore contents of AX
         DEC CH         ; Decrement digit count
         JNZ BACK       ; If not zero repeat

         POP AX         ; Restore registers
         POP CX
         POP DX

         RET
         ENDP
```

Sample Program

```
; Sample program displays 4-digit hex number in AX

.MODEL SMALL
.STACK 100
.CODE

    MOV AX, 12ABH       ; Load AX with test data

    CALL D_HEX          ; Call procedure

    MOV AH, 4CH         ; [Exit
    INT 21H             ;  to DOS]
```

```
D_HEX PROC NEAR

    PUSH DX             ; Save registers
    PUSH CX
    PUSH AX

    MOV CL, 04H         ; Load rotate count
    MOV CH, 04H         ; Load digit count
BACK:   ROL AX, CL      ; Rotate digits
    PUSH AX             ; Save contents of AX
    AND AL, 0FH         ; [Convert
    CMP AL,9            ;  number
    JBE ADD30           ; to
    ADD AL, 37H         ; its
    JMP DISP            ; ASCII
ADD30:  ADD AL,30H      ; equivalent]
DISP:   MOV AH,02H
    MOV DL,AL           ; [Display the
    INT 21H             ;  number]
    POP AX              ; Restore contents of AX
    DEC CH              ; Decrement digit count
    JNZ BACK            ; If not zero repeat

    POP AX              ; Restore registers
    POP CX
    POP DX

    RET
    ENDP
    END
```

```
C:\tasm\tasm s_d_hex.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_d_hex.asm
Error messages:       None
Warning messages:     None
Passes:                 1
Remaining memory:     410k

C:\tasm\tlink s_d_hex.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_d_hex
12AB
```

3.17.5 Lookup Tables for Data Conversions

For certain data conversion, when number of possible data conversions are small in numbers then lookup tables are often used to convert data from one form to another. For example, for conversion of BCD to 7-segment code there are only 10 possible conversions. A lookup table is nothing but a array form in the memory as a list of data that is referenced by a procedure to perform conversions.

Converting from BCD to 7-segment code

Let us see how to perform BCD to 7-segment code conversion. For BCD to 7-segment code conversion a lookup table contains the 7-segment codes for the numbers 0 to 9. These codes are determined from Fig. 3.25. The 7-segment display shown in Fig. 3.25 uses active high (logic 1) input to light a segment. The code is formed by placing the a segment in the bit position 0 and the g segment in the bit position 6. It position 7 is kept 0.

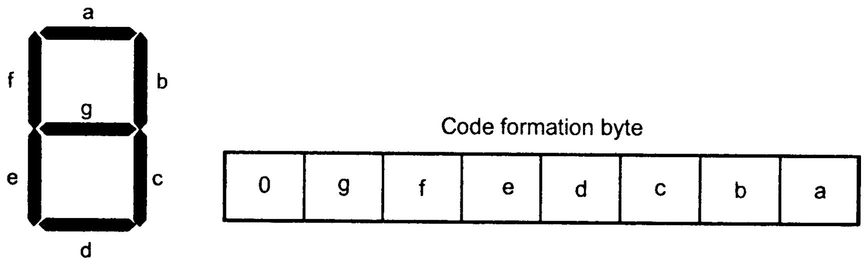


Fig. 3.25 7-segment code formation

A look-up table can be stored in the program memory (code segment) or in the data memory (data segment). Let us see the program which uses lookup table stored in the data memory to convert BCD code into its 7-segment equivalent code.

Program statement : Write an assembly language program to convert BCD to 7-segment code.

Program

```
        .MODEL  SMALL
        .DATA
        TABLE DB  3FH      ; 0
              DB  06H      ; 1
              DB  5BH      ; 2
              DB  4FH      ; 3
              DB  66H      ; 4
              DB  6DH      ; 5
              DB  7DH      ; 6
              DB  07H      ; 7
              DB  7FH      ; 8
              DB  6FH      ; 9
        .CODE
START:     MOV AX, @DATA       ; [Initialize
           MOV DS, AX          ;  Data segment]
           MOV AL, 08H         ; Loads AL with any BCD digit,
                               ; for example 8, to be converted to
                               ; 7-segment code
        MOV BX,OFFSET TABLE    ; Load BX with the offset of
                               ; starting address of lookup table
```

```
        XLAT TABLE              ; Copy byte from address pointed by
                                ; [BX + AL] back into AL
        MOV AH, 4CH             ; [Exit
        INT 21H                 ;  to DOS]
        END START
        END
```

Note : When look-up table is stored in the code segment we have to include a segment override prefix in the XLAT instruction because XLAT instruction by default access, byte from data segment. To access byte from code segment we have modify XLAT instruction as XLAT CS : TABLE.

Look-up table to access ASCII data

Many program require that numeric codes to be converted to ASCII character strings. For example, if we need to display month in the text format we should use lookup table to reference the ASCII coded months of the year. Let us see program to access ASCII string corresponding to given month of the year using look-up table stored in the data segment.

Program statement : Write an assembly language program to access ASCII string corresponding to given month of the year.

Program:

```
        .MODEL SMALL
        .DATA
        DPOINTER DW JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
                 DW OCT, NOV, DIC
            JAN DB       'JANUARY $'
            FEB DB       'FEBRUARY $'
            MAR DB       'MARCH $'
            APR DB       'APRIL $'
            MAY DB       'MAY $'
            JUN DB       'JUNE $'
            JUL DB       'JULY $'
            AUG DB       'AUGUST $'
            SEP DB       'SEPTEMBER $'
            OCT DB       'OCTOBER $'
            NOV DB       'NOVEMBER $'
            DIC DB       'DECEMBER $'
        .CODE
START:      MOV AX,@ DATA       ; [Initialize
            MOV DS, AX          ;  Data segment]
            MOV AL, 07H         ; Loads AL with any month in its
                                  numerical value
      MOV SI, OFFSET DPOINTER   ; Address table find month of year
      MOV AH,00H                ; [Multiply the AL by 2
      ADD AX, AX                ;  to point to correct
      ADD SI, AX                ;  month of the year]
      MOV DX, [SI]              ;  Get month of year
      MOV AH, 09H               ; [Display month
```

```
INT 21H             ;  of year string]
MOV AH,4CH          ; [Exit
INT 21H             ;  to DOS]
END START
END
```

3.18 Procedures

Whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them. One way is to write the group of instructions as a separate procedure. We can then just CALL the procedure whenever we need to execute that group of instructions. For calling the procedure we have to store the return address onto the stack. This process takes some time. If the group of instructions is big enough then this overhead time is negligible with respect to execution time. But if the group of instructions is too short, the overhead time and execution time are comparable. In such cases, it is not desirable to write procedures. For these cases, we can use macros. Macro is also a group of instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a procedure. The disadvantage of macro is that it generates inline code each time when the macro is called which takes more memory. In this section we discuss the procedures.

From the above discussions, we know that the procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called **near procedure** otherwise it is referred to as **far procedure**. For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program and procedure. But for far procedures CALL instruction pushes both IP and CS on the stack. Let us see the detail description and examples of CALL instruction to enter the procedure and RET instruction to return from the procedure.

CALL Instruction :

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A **far CALL** is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

Examples :

Direct within segment (near)

```
CALL PRO              ; PRO is the name of the procedure.
                      ; The assembler determines displacement of pro
                      ; from the instruction after the CALL and codes
                      ; this displacement in as part of the instruction.
```

Indirect within-segment (near)

```
CALL CX               ; CX contains, the offset of the first instruction
                      ; of the procedure. Replaces contents of IP with
                      ; contents of register CX.
```

Indirect to another segment (far)

```
CALL DWORD PTR [BX]   ; New values for CS and IP are fetched from four
                      ; memory locations in DS. The new value for CS
                      ; is fetched from [BX] and [BX + 1], the new IP
                      ; is fetched from [BX + 2] and [BX + 3].
```

RET Instruction :

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

A RET instruction can be followed by a number, for example, RET 4. In this case the stack pointer will be incremented by an additional four addresses after the IP or the IP and CS are popped off the stack. This form is used to increment the stack pointer up over parameters passed to the procedure on the stack.

Flags : The RET instruction affects no flags.

3.18.1 Reentrant Procedure

In some situations it may happen that procedure1 is called from main program, procedure2 is called from procedure1 and procedure1 is again called from procedure2. In this situation program execution flow reenters in the procedure1. This type of procedures are called **reentrant procedures**. The flow of program execution for reentrant procedure is shown in Fig. 3.26.

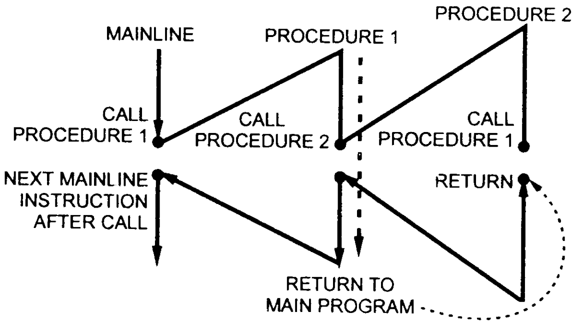
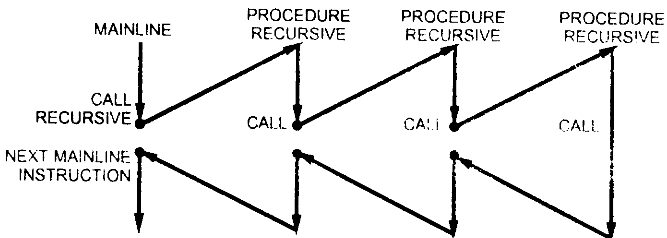


Fig. 3.26 Flow of program execution for reentrant procedure

3.18.2 Recursive Procedure

A recursive procedure is a procedure which calls itself. Recursive procedures are used to work with complex data structures called trees. If the procedures is called with N (recursion depth) = 3. Then the n is decremented by one after each procedure CALL and the procedure is called until n = 0. Fig. 3.27 shows the flow diagram and pseudo-code for recursive procedure.




```
PROCEDURE RECURSIVE
IF N≠0
    DECREMENT N
    CALL RECURSIVE
ELSE
    RETURN
```

Fig. 3.27 Flow diagram and pseudo-code for recursive procedure

3.19 Macro

Macro is a group of instructions. The macro assembler generates the code in the program each time where the macro is 'called'. Macros can be defined by MACRO and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can be used in the program, as shown below.

Example : Macro definition for initialization of segment registers.

```
INIT MACRO          ; Define macro
MOV AX, @ data      ;  ┐
MOV DS              ;  } Body of macro definition
MOV ES, AX          ;  ┘
ENDM                ; End macro
```

It is important to note that macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as **Macro expansion.**

Comparison of Procedure and Macro

Sr. No.	Procedure	Macro
1.	Accessed by CALL and RET instruction during program execution.	Accessed during assembly with name given to macro when defined.
2.	Machine code for instructions is put only once in the memory.	Machine code is generated for instructions each time when macro is called.
3.	With procedures less memory is required.	With macros more memory is required.
4.	Parameters can be passed in registers, memory locations, or stack.	Parameters passed as part of statement which calls macro.

Table 3.8

Passing Parameters in Macro

In Macro, parameters are passed as a part of statement which calls Macro.

Example :

```
PROMPT MACRO MESSAGE ;Define macro with MESSAGE as a parameter
       MOV AH, 09H
       LEA MESSAGE
       INT 21H
       ENDM                  ;End macro
     . DATA
       MES1     DB 10, 13, 'Student Name : $'
       MES2     DB 10, 13, 'Student Address : $'
    .CODE
START: MOV AX, @data             ; [ Initialize
       MOV DS, AX                ;   data segment ]
```

```
PROMPT MES1                  ; Display MES1
PROMPT MES2                  ; Display MES2
MOV AH,4CH                   ; Return to DOS
INT 21H
END START
```

The above example shows that parameters can be passed in macro with the help of dummy argument. Argument tells the assembler to match its name with any occurrence of the same name in the macro body. For example the dummy argument MESSAGE also occurs in the LEA instruction. The macro instruction "PROMPT MES1" passes the MES1 as a parameter and macro accepts that as an argument.

Local Variables in a Macro

Body of the Macro can use local variables. A local variable defined in the Macro is available in the Macro, however it is not available outside the Macro. To define a local variable, LOCAL directive is used. Example shows how local variable is used as a jump address. If this jump address is not defined as a local, the assembler give an error message on the second and subsequent attempts to use the Macro.

Example

```
DISPLAY  MACRO A            ; Displays ASCII character in uppercase
         LOCAL J_LABEL;     Defines J_LABEL as local
         PUSH DX
         CMP AL,'Z'
         JBE J_LABEL        ; Check if uppercase
         SUB AL,20H         ; Convert to uppercase
J_LABEL: MOV DL,AL
         MOV AH,02H
         INT 21H
         POP DX
         ENDM
```

The above Macro accepts ASCII code for character. (A-Z or a-z). If it is for lowercase character, Macro converts it to uppercase character and displays the uppercase character on video screen.

It is important to note that local variable or variables must be defined using LOCAL directive immediately after MACRO directive.

3.20 Instruction Formats

The instructions of 8086 vary from 1 to 6 bytes in length. Fig. 3.28 shows the instruction formats for 1 to 6 bytes instruction for each instruction format first field is the operation code field, commonly known as opcode field. Opcode field indicates the type of operation to be performed by the processor. The other field in the instruction format is operand field. The operand field may consists of source/destination operand, source operand address, destination operand address or next instruction address. The operand and the relative address of the operand (displacement) may be either 8-bit or 16-bit long depend on the instruction and its addressing mode.

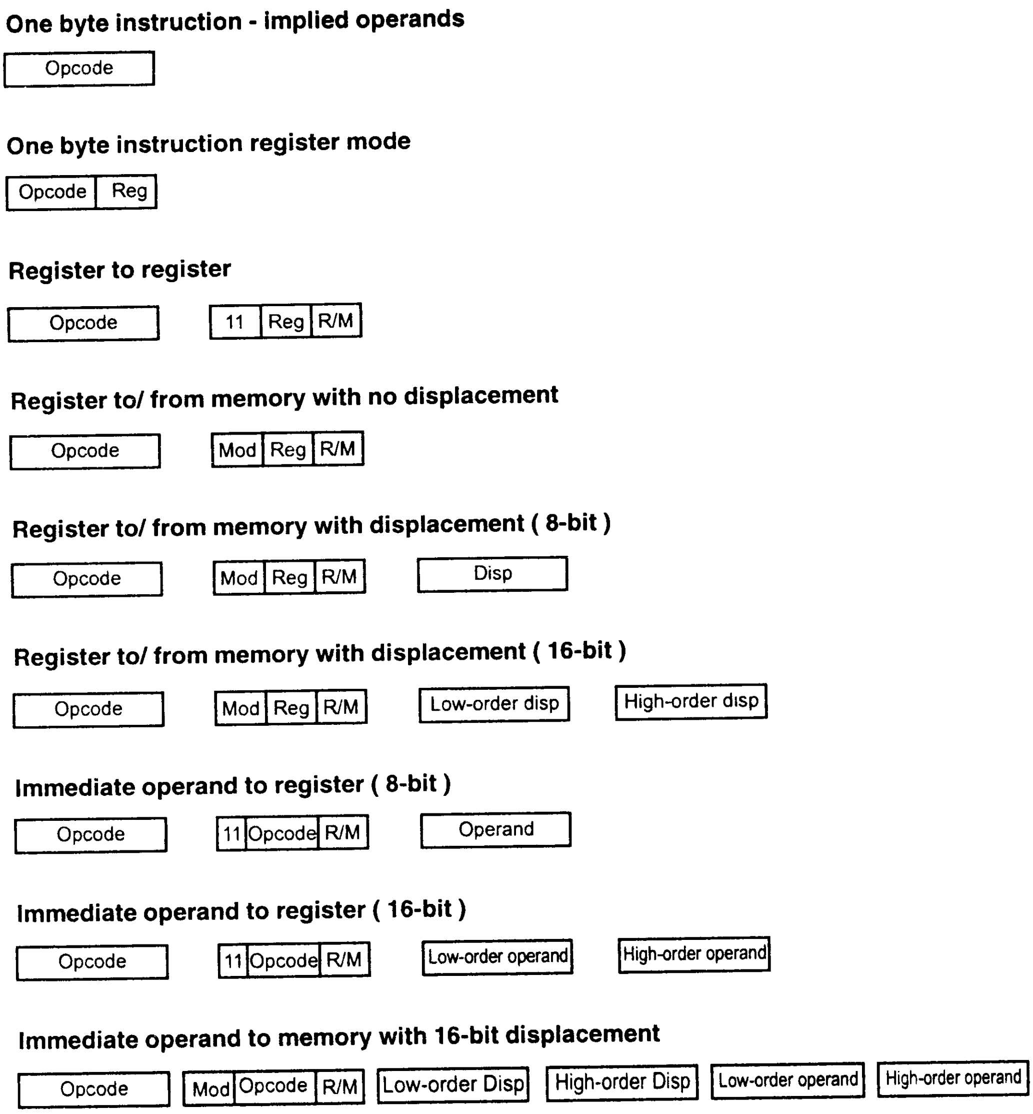


Fig. 3.28 Sample 8086 instruction formats

The opcode and the addressing mode is specified using first two bytes of an instruction. The opcode/addressing mode byte(s).

The opcode/addressing mode byte(s) may be followed by :

- No additional byte.
- Two byte EA (For direct addressing only).
- One or two byte displacement.
- One or two byte immediate operand.
- One or two byte displacement followed by a one or two byte immediate operand.

Two byte displacement and a two byte segment address (for direct intersegment addressing only).

Most of the opcodes in 8086 has a special 1-bit indicators. They are :

W-bit : Some instructions of 8086 can operate on byte or a word. The W-bit in the opcode of such instruction specify whether instruction is a byte instruction (W = 0) or a word instruction (W = 1).

D-bit : The D-bit in the opcode of the instruction indicates that the register specified within the instruction is a source register (D = 0) or destination register (D = 1).

S-bit : An 8-bit 2's complement number can be extended to a 16-bit 2's complement number by making all of the bits in the higher-order byte equal the most significant bit in the low order byte. This is known as sign extension. The S-bit along with the W-bit indicate :

S	W	Operation
0	0	8-bit operation
0	1	16-bit operation with 16-bit immediate operand
1	0	–
1	1	16-bit operation with a sign extended 8-bit immediate operand

Table 3.9

V-bit : V-bit decides the number of shifts for rotate and shift instructions. If V = 0, then count = 1; if V = 1, the count is in CL register. For example, if V = 1 and CL = 2 then shift or rotate instruction shifts or rotates 2-bits.

Z-bit : It is used for string primitives such as REP for comparison with ZF Flag. If it is 1, the instruction with REP prefix is executed until the zero flag matches the Z-bit.

(Refer Appendix A for instruction formats)

As seen from the Fig. 3.28 if an instruction has two opcode/addressing mode bytes, then the second byte is of one of the following two forms :

MOD	Opcode	R/M

or

MOD	Reg	R/M

where Mod, Reg and R/M fields specify operand as described in the following tables.

Mode		Displacement
0	0	Disp = 0 Low order and High order displacement are absent
0	1	Only Low order displacement is present with sign extended to 16-bits.
1	0	Both Low-order and High-order displacements are present.
1	1	r/m field is treated as a 'Reg' field.

Table 3.10 'Mod' field assignments

Word Operand (W = 1)		Byte Operand (W = 0)		Segment	
0 0 0	AX	0 0 0	AL	0 0	ES
0 0 1	CX	0 0 1	CL	0 1	CS
0 1 0	DX	0 1 0	DL	1 0	SS
0 1 1	BX	0 1 1	BL	1 1	DS
1 0 0	SP	1 0 0	AH		
1 0 1	BP	1 0 1	CH		
1 1 0	SI	1 1 0	DH		
1 1 1	DI	1 1 1	BH		

Table 3.11 'Reg' field assignment

R/M	Operand Address
0 0 0	EA = [BX] + [SI] + Displacement (optional)
0 0 1	EA = [BX] + [DI] + Displacement (optional)
0 1 0	EA = [BP] + (SI) + Displacement (optional)
0 1 1	EA = [BP] + [DI] + Displacement (optional)
1 0 0	EA = [SI] + Displacement (optional)
1 0 1	EA = [DI] + Displacement (optional)
1 1 0	EA = [BP] + Displacement (optional)
1 1 1	EA = [BX] + Displacement (optional)

Table 3.12 'R/M' field assignment

Example 3.4 : *Write the instruction format for PUSH BX instruction.*

Solution : This instruction will put BX register contents on stack. Referring the table in Appendix A we find that the 5-bit opcode for this instruction is 01010. We put 011 in the REG field to represent the BX register. The codes for each registers are shown in Table 2.11. The resultant code for PUSH BX will be 01010011.

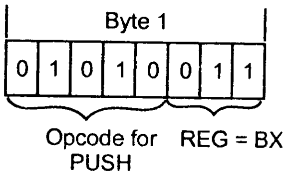


Fig 3.29 Instruction format for PUSH BX

Example 3.5 : *Write the instruction format for MOV AX, CX instruction.*

Solution : This instruction will copy a word from the CX register to the AX register. Referring the table in Appendix A we find the 6-bit opcode for this instruction is 100010. Because we are moving a word, W=1. The D bit for this instruction may be somewhat confusing . Since two registers are involved, we can think of the move as either to AX or from CX. It actually does not matter which we assume as long as we are consistent in coding the rest of the instruction. If we think of the instruction as moving a word to AX, then make D=1 and put 000 in the REG field to represent the AX register. The MOD field will be 11 to represent register addressing mode. We make the R/M field 001 to represent the other register CX. The resultant code for the instruction MOV AX, CX will be 10001011 11000001. The Fig 3.30 shows the meaning of all these bits.

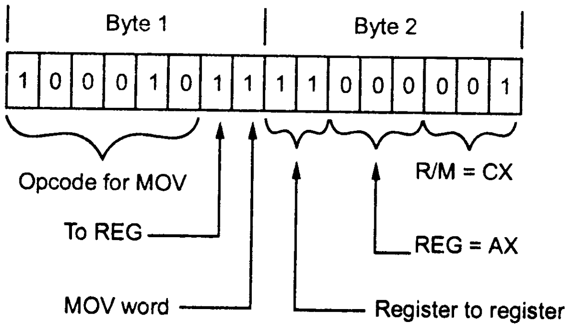


Fig. 3.30 Instruction format for MOV AX, CX

If we change D field to a 0 and swap the codes in the REG and R/M field, we will get 10001001 11001000, which is another equally valid code for the instruction.

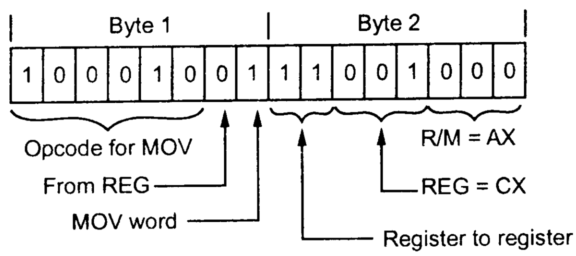


Fig. 3.31 Alternative instruction format for MOV AX, CS

Example 3.6 : *Write the instruction format for MOV 56H[SI], BH*

Solution : This instruction will copy a byte from the BH register to a memory location. The BIU will compute the effective address of the memory location by adding the indicated displacement of 56H to the contents of SI register. The BIU then produce the physical address by adding the effective address with the base represented by 16-bit contents of DS register. The 6-bit opcode for this instruction is again 100010. We put 111 in the REG field to represent the BH register. D = 0 because we are moving data from BH register. W = 0 because we are moving a byte. The R/M field will be 100 because SI contains part of the effective address. The MOD field will be 01 because the displacement contained in the instruction, 56H, will fit in 1 byte. The 8-bit displacement forms the third byte of the instruction. The resultant sequence of code bytes will be 10001000 01111100 01010110.

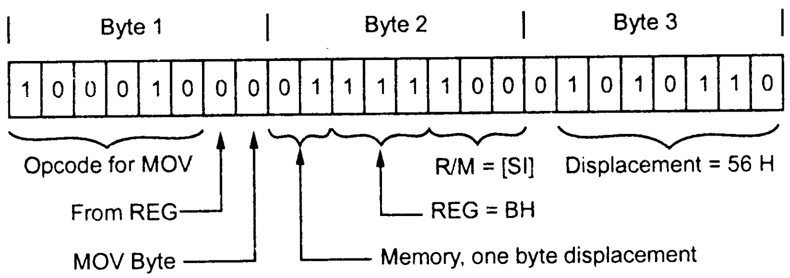


Fig. 3.32 Instruction format for MOV 56H [SI], BH

Example 3.7 : *Write the instruction format for MOV DL, [BX].*

Solution : This instruction will copy a byte to DL from the memory location whose effective address is contained in BX. The effective address will be added to the data segment base in DS to produce the physical address. Referring the table in Appendix A, we find opcode for this instruction is 100010. We make D = 1 because data is being moved to register DL. We make W = 0 because the instruction is moving a byte into DL. We put 010 in REG field to represent DL register. We make MOD field 00 to represent memory with no displacement. For this instruction R/M field will be 111. The resultant sequence of code bytes will be 1000101000010111.

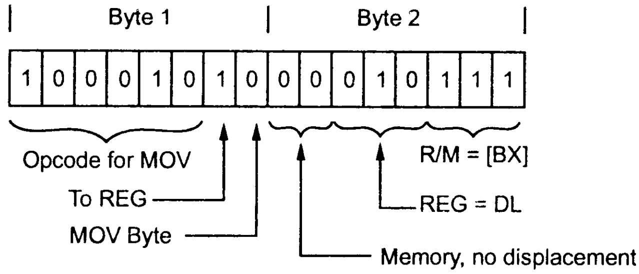


Fig. 3.33 Instruction format for MOV DL, [BX]

Example 3.8 : *Write the instruction format for MOV BX, [1234H]*

Solution : This instruction copies the contents of two memory locations into the BX register. The direct address or displacement of the first memory location from the start of the data segment is 1234H. The BIU will produce the physical memory address by adding this displacement to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. We make D = 1 because we are moving data to the BX register, and we make W = 1 because the data being moved is a word. We put 011 in the REG field to represent the BX register. Referring Tables 3.11 and 3.12 we get MOD = 00 and R/M field = 110. Then the first two bytes of instruction code will be 10001011 00011110. These two bytes will be followed by the low byte of the direct address, 34H (0011 0100 binary), and the high byte of the direct address, 12H (0001 0010 binary). The instruction will be coded into four successive memory addresses as 8BH, 1EH, 34H and 12H.

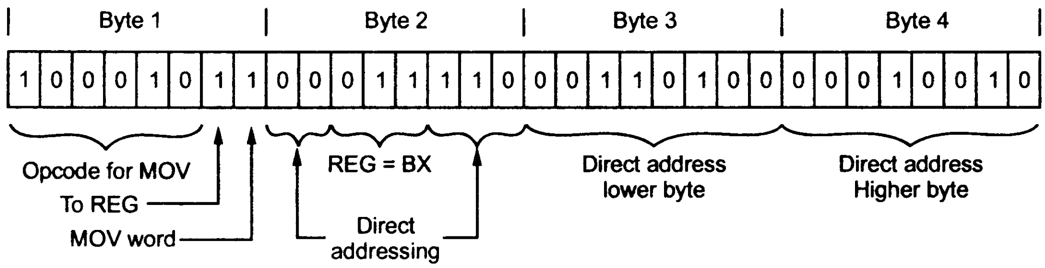


Fig. 3.34 Instruction format for MOV BX, [1234H]

Example 3.9 : *Write the instruction format for MOV CS : [BX], CL.*

Solution : This instruction copies a byte from the CL register to a memory location. The effective address for the memory location is contained in the BX register. Usually an effective address in BX will be added to the data segment base in DS to produce the physical memory address. In this instruction, the CS in front of [BX] indicates that we want the BIU to add the effective address to the code segment base in CS to produce the physical address. The CS : is called segment override prefix.

When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put in memory before the code for the instruction. The code byte for the segment override prefix has the format 001 XX 110. We can be replace XX with : the segment code. The segment codes are : ES = 00, CS = 01, SS = 10 and DS = 11. The segment override prefix byte for CS, then, is 00101110.

The opcode for this instruction is 100010. D = 0 because we are moving data from the CL register. W = 0 because we are moving a byte. We put 001 in REG field to represent CL register. We make MOD field 00 to represent memory with no displacement. For this instruction R/M field will be 111. The resultant sequence of code bytes will be 00101110 10001000 00001111.

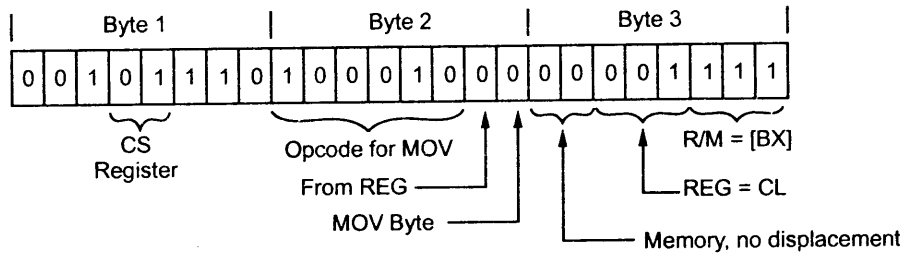


Fig. 3.35 Instruction format for MOV CS : [BX], CL

Review Questions

1. *Explain various data addressing modes of 8086 with the help of examples.*
2. *Explain the difference between direct and indirect addressing mode.*
3. *Explain base-plus-index addressing mode.*
4. *Explain how base-plus-index addressing mode can be used to locate array data.*
5. *Explain register relating addressing.*
6. *Explain base relative-plus-index addressing.*
7. *Explain how base relative-plus-index addressing can be used to locate data from two dimensional array.*
8. *Explain the string addressing mode.*
9. *Explain various I/O addressing modes supported by 8086.*

10. *Explain direct program memory addressing with the help of example.*
11. *What is short, near and far jumps ?*
12. *Explain the difference between intersegment and intrasegment jump instructions.*
13. *Explain relative program memory addressing.*
14. *Explain indirect program memory addressing.*
15. *What is stack ?*
16. *What is the function of stack pointer ?*
17. *What do you mean by top of stack ?*
18. *Explain the usefulness of the following instructions in 8086*

    a. LOCK b. TEST c. XLAT d. LES.
19. *Write the difference between the following instructions*

    a. MOV CX, 437AH and MOV CX, [437AH]

    b. MOV BL, 437AH and MOV BL, DS:BYTE PTR [437AH].
20. *Can we write following instructions for microprocessor 8086 ?*

    a. MOV CX, AL b. MOV DS, 437AH

    c. MOV CL, [BX] d. MOV 43H[SI], DH

    e. MOV CS:[BX], DL.
21. *With the help of an example describe the action performed by microprocessor 8086 for each of the following instructions :*

    a. AAM b. CMPSB c. IMUL d. ROL.
22. *Explain the use of the following prefixes*

    a. REP b. REPE.
23. *Describe the response of 8086 to the following five primitive string operations. MOVS, CMPS, SCAS, LODS and STOS*
24. *Discuss all types of jump instructions used in 8086 microprocessor.*
25. *Write a operations performed by the 8086 microprocessor CALL instruction.*
26. *Explain in detail the difference between near CALL and far CALL.*
27. *For the following instruction compute the address of memory operand for 8086 :*

    a. MOV AX, [BX] b. MOV AL, [BP + SI]

    Assume :

    CS = 0100H DS = 0200H SS = 0400H ES = 0030H

    BP = 0010H DX = 0020H SI = 0030H SP = 0030H

    Clearly show computations.
28. *Describe the difference between a jump and a call instruction ? What does the processor do in executing it ? You may use 8085, 8086 instructions to explain.*
29. *Explain what operation is performed by the following instructions :*

    a. SHL BYTE PTR [0400 H], CL

    b. MOV [BX] [DI] + 4, AX

    c. XLAT d. XTHL e. PCHL.

30. *Explain the use of PUSH and POP instructions in 8086.*
31. *Explain the function of the following instructions of 8086 :*
    XLAT, CWD and CMPSB.
32. *What is the function of assembler directives ?*
33. *Explain the following assembler directives*
    a. DB b. EXTRN c. .MODEL SMALL d. PROC e. PUBLIC.
34. *Explain variables, suffix and operators used in assembly language programming.*
35. *What do you mean by machine language program ?*
36. *What do you mean by assembly language program ?*
37. *Give the difference between machine language and assembly language.*
38. *Explain the assembly language programming tips.*
39. *What do you mean by optimum solution ?*
40. *Explain the steps that assembler follows to convert .ASM file to .OBJ file.*
41. *Explain the function of linker.*
42. *What is debugger ? Explain its advantages.*
43. *Explain various debugger commands.*
44. *What is time delay ? Write an assembly language program to generate a delay of 500 ms.*
45. *Explain the two techniques to convert binary to ASCII.*
46. *Explain the process of converting ASCII to binary.*
47. *Explain the process of displaying hexadecimal data.*
48. *Explain how look up tables can be used to convert BCD to 7-segment code.*
49. *What is macro ? When it should be used ? What are its advantages ?*
50. *Explain the structure of macro with the help of example.*
51. *Give the comparison between procedure and macro.*
52. *How are parameters passed to a macro ?*

❑❑❑

4 BIOS and DOS Interrupts

In the previous chapters we have seen various hardware components of the microcomputer system. The hardware on its own is of no use (in the sense that we can't write or run programs on it) and requires software utilities to make it usable. These are as follows

1. A permanent loader program that is executed when the power is switched ON.
2. A program that initializes and contains drivers for all interfaces.
3. A program that will load and execute other programs.
4. A program that will handle logical files.
5. Programs to implement special features of the system such as :
   - Time of day clock and
   - Graphics mode initialization.

The above programs perform resource management and serve as an interface between the user and the hardware. These programs are collectively known as the **operating system**.

In IBM PC, part of the operating system is located in the permanent memory (ROM) and part is loaded during power up. The part located in ROM is referred to as **ROM-BIOS** (Basic Input/Output System). The other part which is loaded in RAM during power-up from hard disk or floppy disk is known as **DOS** (Disk Operating System).

4.1 ROM-BIOS (Basic Input/Output System)

BIOS is located in an 8 kbyte ROM at the top of memory, the address range being from FE000H to FFFFFH. The programs within ROM-BIOS provide the most direct, lowest level interaction with the various devices in the system. The ROM-BIOS contains routines for

1. Power-on self test
2. System configuration analysis

3. Time-of-day
4. Print screen
5. Bootstrap loader
6. I/O support program for
    a. Asynchronous communication
    b. Keyboard
    c. Diskette
    d. Printer
    e. Display.

Most of these programs are accessible to the assembly-language programmer through the software interrupt instruction (INT). The design goal for the ROM-BIOS programs is to provide a device-independent interface to the various physical devices in the system. The following section describes what is meant by device-independent interface to the various physical devices in the system.

Let us see the parallel printer interface as an example :

```
OUT_CHAR :    IN AL, STATUS          ; Get status of printer
              TEST AL, 01H           ; Is it busy
              JNZ OUT_CHAR           ; Yes, try again
              MOV AL, CHAR           ; Get character
              MOV DX, ADDR DATA      ; Get address
              OUT DX, AL             ; Send character
```

To run this program successfully, it is necessary to know the physical address of status and ADDR DATA (Address of data port). It is also necessary to know the location and desired state of the "BUSY BIT". Now, we will see same program with BIOS CALL.

Program with ROM-BIOS CALL

```
OUT_CHAR :    MOV AL, CHAR           ; Get character
              MOV AH, 00H            ; Function 0 = output
              INT 17H                ; Send to BIOS routine
```

In the above program, AL and AH hold the character to be printed and function number respectively. It is absolutely not necessary to know anything about the hardware. So we can say that the later program is device/hardware independent program and interface is device/hardware independent interface.

4.2 Disk Operating System (DOS)

It is seen that ROM-BIOS provides basic low-level services. Using ROM-BIOS one can output characters to various physical devices like the printer or the display monitor, one can read characters from keyboard, one can read or write sectors of data to the diskette. But still few things we cannot do with ROM-BIOS.

1. It is not possible to provide ability to load and execute programs directly.
2. It is not possible to store data on the diskette organized as logical files.

3. ROM-BIOS has no command-interpreter to allow us to copy files, print files, delete files.

It is DOS that provides these services. When we turn our computer ON, we expect to see a message or a prompt. We except to be able to look at the diskette directory to see what data files or programs the diskette contains. We expect to run a program by typing its name. We want to copy programs from one diskette to another, print programs, and delete programs. All these services are provided by group of programs called DOS. The services provided by DOS can be grouped into following categories.

1. Character device I/O : This group includes routines that input or output characters to character oriented devices such as the printer, the display monitor, and the keyboard.

2. File management : This group includes routines that manage logical files, allowing you to create, read, write and delete files.

3. Memory management : This group includes routines that allow us to change, allocate, and deallocate memory.

4. Directory management : This group includes routines that permit us to create, change search, and delete directories.

5. Executive functions : This group includes routines that allow us to load and execute programs, to overlay programs, to retrieve error codes from completed programs, and to execute commands.

6. Command interpreter : This routine is in action whenever a prompt is present on the screen. It interprets commands and executes DOS functions, utility programs, application programs, depending upon the command.

7. Utility programs : These programs facility to copy, delete provides the DISKCOPY, DIR and many other DOS commands.

Comparison between DOS and ROM-BIOS

Sr. No.	DOS	BIOS
1.	DOS is loaded from the bootable diskette.	BIOS is located in an 8 kbyte ROM.
2.	DOS program offer different degree of flexibility, portability, and hardware independence.	The programs within the ROM-BIOS provide the most direct, lowest level interaction with the various devices in the system. Using these programs require hardware knowledge.
3.	DOS has ability to load and execute programs directly.	ROM-BIOS does not have ability to load and execute programs directly.
4.	DOS can store data on the diskette organized as a logical files.	ROM-BIOS cannot store data on the diskette organized as a logical files.
5.	DOS has a command-interpreter to allow us to copy files, print files and delete files.	ROM-BIOS has no command-interpreter to allow us to copy files, print files, and delete files.

4.2.1 Intervals of DOS

We have seen that DOS is not located in the ROM with ROM-BIOS. It is stored on a diskette and is loaded into RAM by a bootstrap loader in BIOS. DOS is distributed into four parts as shown in Fig. 4.1.

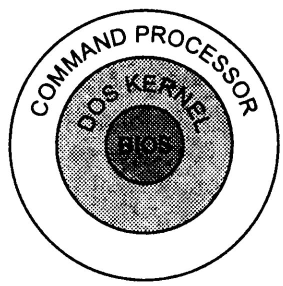


Fig. 4.1 Structure of DOS

1. Bootstrap loader

Bootstrap loader is a combination of ROM bootstrap routine and disk bootstrap routine. The ROM bootstrap routine is not the physical part of DOS, but it is a logical extension to DOS, since it is a part of ROM-BIOS. These two routines are used to load DOS into the memory.

2. BIOS

The BIOS is the inner level of the DOS (Don't confuse BIOS with ROM BIOS). It is provided by the manufacturer of the system. It contains the default resident hardware dependent drivers for

1. Console display and keyboard (CON)
2. Line printer (PRN)
3. Auxiliary device (AUX)
4. Date and time (CLOCK $)
5. Boot disk device (block device).

The BIOS is read into RAM during system initialization as part of a file named IO.SYS. (In PC-DOS, the file is called IBMBIO.COM). It is responsible for determining equipment status, initializing equipment, and loading device drivers.

3. DOS Kernel

The DOS Kernel is read into memory during system initialization from the MSDOS.SYS file on the boot disk (The file is called IBMDOS.COM in PC DOS). It provides interface between DOS and user programs through the DOS function calls.

4. Command processor or command interpreter

The command processor or command interpreter is an outer layer called **shell**. It is an user interface to the operating system. The command processor or command interpreter is responsible for parsing and carrying out user commands. The command processor or command interpreter is available in the file named COMMAND.COM. It is divided into three parts.

1. A resident portion
2. An initialization section
3. A transient section

Resident portion : Resident Portion of the COMMAND.COM is responsible for processing routines for Ctrl-C, Ctrl-Break, critical errors, and the termination of other transient programs. The resident portion also contains the program required to reload the transient portion of the COMMAND.COM if necessary.

Initialization section : Initialization portion of the COMMAND.COM processes the AUTOEXEC.BAT file, which executes the list of commands at system startup.

Transient section : The transient section issues user prompt, reads the commands from the keyboard for batch file, and executes these commands. The user commands are divided into three categories.

1. Internal commands (COPY, REN, DIR, etc.)
2. External commands
3. Batch files.

The code for the internal commands is embedded in the COMMAND.COM. But the code for external commands must be loaded from the disk into the **Transient Program Area** (TPA) of the memory before execution of the external command. The external commands are the executable program files with the extensions .EXE, .COM or .BAT. The MS-DOS uses the EXEC function to load and execute these external commands.

4.2.2 Loading of DOS

When the system is started or reset, program execution begins at address 0FFFF0H. The addresses 0FFFFH lies within an area of ROM and it contains a jump instruction to transfer control to system test code, Power On Self Test (POST). Then the control is transferred to the ROM bootstrap routine. The ROM bootstrap routine reads the disk bootstrap routine from the first sector of the system startup disk (the boot sector) into memory at some arbitrary address and then transfers control to it.

In a PC-XT or AT, the machine may require to boot from the winchester in drive C rather than from a floppy diskette in drive A. In this case, there should be no diskette in drive A. The ROM program senses the absence of a diskette in drive A and tries to load the disk bootstrap program from drive C.

The diskbootstrap checks to see if the 'boot' disk contains DOS by checking the first sector of the root directory for the file IO.SYS and MSDOS.SYS (These are referred to as IBMBIO.COM and IBMDOS.COM in PC-DOS). If these are not found in the bootdisk, the user is prompted to change disks and strikes any key to try again. If the two files are found, the diskbootstrap reads them into memory and transfers the control to the IO.SYS.

The IO.SYS file consists of two separate modules. The first is the BIOS, which contains the linked set of resident device drivers for the console, auxiliary port, printer, clock devices, and some hardware specific initialization code. Second module consists of system initialization program (SYSINIT) which determines the RAM size in the PC and based on this information moves itself to high memory. Then, it loads the MSDOS.SYS (or

IBMDOS.COM) program to its final memory location or shifts it from its original load location to the final one. The final location of the DOS Kernel program, MSDOS.SYS, may actually overwrite the now unnecessary portions of IO.SYS. This sequence finally ends with control being transferred to MSDOS.SYS.

The DOS Kernel initializes its tables and sets up its various work areas. It sets up the interrupt vectors for the DOS interrupts 20H-2FH pointing them to appropriate service routines (which are also a part of DOS). It then loads and executes the device drivers. These driver functions determine the equipment status, perform necessary hardware initialization, and set up vectors for any external hardware interrupts. It allocates appropriate buffers, e.g. for disk, and finally returns control to the system initialization program (SYSINIT).

After the initialization of DOS Kernel and all device drivers are available, SYSINIT calls the normal MS-DOS file service to open the CONFIG.SYS file. The CONFIG.SYS contains a list of additional device drivers that the user wants in his system. Typical example of such additional drivers are VDISK.SYS to establish a RAM-disk (i.e. an area of memory that behaves exactly like a disk) or ANSI.SYS to load the expanded keyboard and screen control programs. This program can also change the default values of the number of files that DOS can simultaneously keep open and the number of buffers allocated for files. An example of CONFIG. SYS is shown below.

```
BUFFERS = 30
  FILES = 20
 DEVICE = ANSI.SYS
```

The additional drivers indicated in the CONFIG.SYS file are sequentially loaded into memory, initialized by calls to their INIT modules, and linked into the device-driver list. The INIT function of each driver tells SYSINIT how much memory to reserve for that driver.

After loading of all installable device drivers, SYSINIT closes all file handles and reopens the console (CON), printer (PRN), and auxiliary (AUX) devices as the standard input, standard output, standard error, standard list, and standard auxiliary devices. This allows a user installed character device-driver to override the BIOS's resident drivers for the standard devices.

Finally, SYSINIT calls the EXEC function to load the command interpreter, or shell. (The default interpreter can be substituted by means of the CONFIG.SYS as mentioned earlier). Once the interpreter is loaded, it displays a prompt and waits for the user to enter a command. Fig. 4.2 shows the DOS memory map and how different components of DOS share the system memory.

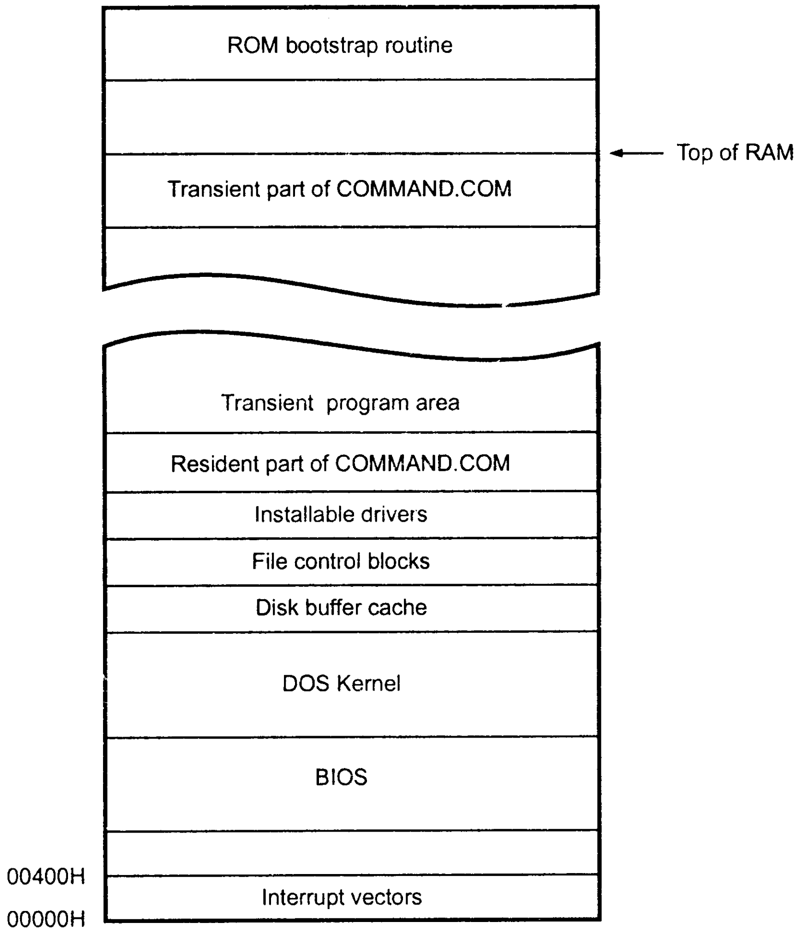


Fig. 4.2 DOS memory map

4.3 Executable Files

The programs that are executed by the PC, operating under DOS, are of two types :

- Programs with .EXE extension and
- Programs with .COM extension.

The one main difference between these two programs is that .COM programs use only one segment, while .EXE programs use many segments. Therefore, .COM programs can have a maximum size of approximately 64 kB and .EXE program can be as large as available memory. The .COM programs fit in the tiny model, in which all segment registers contain the same value; that is, the code and data are mixed together. In contrast, .EXE programs fit in the small, medium or large model, in which the segment registers contain different values; that is, the code, data and stack reside in separate segments. The

.EXE programs can have multiple code and data segments. Let us see the structure of these programs in detail.

4.3.1 Introduction to .COM Programs

A .COM program resides on the disk as an absolute image of the machine instructions to be executed. Because .COM program does not contain relocation information, they are more compact, and are loaded for execution slightly faster than .EXE programs.

The .COM programs are loaded immediately above the program segment prefix (PSP) and they do not have header to specify entry point. The entry point or origin of .COM program is 0100H. It is the length of the PSP. The location 0100H contains an executable instruction in the .COM program. The maximum length of a .COM program is 65,536 bytes (one segment length) minus the length of the PSP (256 bytes) and a mandatory word of stack (2 bytes). Because the .COM programs use only one segment, all jump and call instructions in the .COM program will be of NEAR type.

When control is transferred to the .COM program from MS-DOS, all of the segment registers point to the PSP. The stack point (SP) register contains 0FFFEH if memory allows; otherwise, it is set as high as possible in memory minus 2 bytes. This is illustrated in Fig. 4.3.

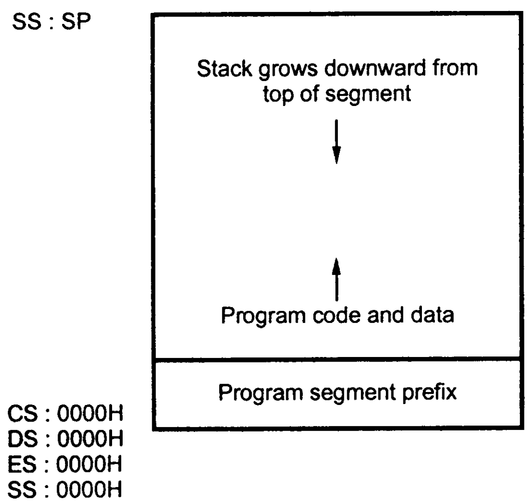


Fig. 4.3 A memory image of a typical .COM program after loading

When .COM program finishes executing, it can return control to MS-DOS by several means. The preferred method is INT21H function 4CH.

Sample .COM Program

The .COM program given below displays WELCOME message on the screen. Only one segment is used in the program. The statement ORG 100H takes care of entry point.

Remember that the entry point should be at 100H. The ASSUME statement in this program tells the assembler that CS and DS segment registers are going to use to point to the code and data segments. But both the segment registers are initialized before entry to point to the PSP.

```
PAGE 50,132
TITLE    WELCOME
; Writes WELCOME on the screen
code segment
         Assume CS:    code,    DS : code
         ORG 100H            ; Initialization of entry point
Start:   MOV AH,09H
         MOV DX,OFFSET MES
         INT 21h             ; [Call DOS function
                             ; to display message]
         MOV AH,4CH
         INT 21H             ; [Call DOS function to terminate
                             ; the program]
; data area
         MES DB 'WELCOME.$'
         Code ENDS
```

4.3.2 Introduction to .EXE Program

As mentioned earlier, .EXE programs can have many logical segments, i.e. more than one code segment, data segment or stack segment. At the time of assembling, assembler does not know where the data segment will be. So it is not possible to assign address to the data segment. The actual location of the data segment is determined when the program is loaded in the memory. After loading of program the actual addresses are known and can be assigned. This process of assigning the actual/physical addresses is called **relocation**. There are many items in .EXE programs need relocation when the program is loaded. The information about items that needing relocation is kept in the file itself, in the **file header**. When .EXE program is loaded in the memory, DOS refers to the file header to find the items that need relocation. The size of this header varies according to the number of program instructions that need to be relocated at load time, but it is always a multiple of 512 bytes. Due to the file header .EXE program is larger than corresponding .COM program and .EXE programs take longer time because of the relocation process.

Before MS-DOS transfers control to the program, it performs following steps :

- Reads the formatted part of the file header into memory.
- Calculates the size of the executable module and reads the module into memory at the start segment.
- Reads the relocation table items into a work area and adds the value of each item to the start segment value.
- Sets the DS and ES registers to the segment address of the PSP.

- Sets the SS register to the address of PSP, plus 100H (the size of the PSP), plus the SS offset value stored in the file header. Also, sets the SP register to the value mentioned in the file header.
- Sets the CS to the address of the PSP, plus 100H (the size of the PSP), plus the CS offset value in the header. Also sets the IP with the offset value mentioned in the file header. The Fig. 4.4 shows this initialization.

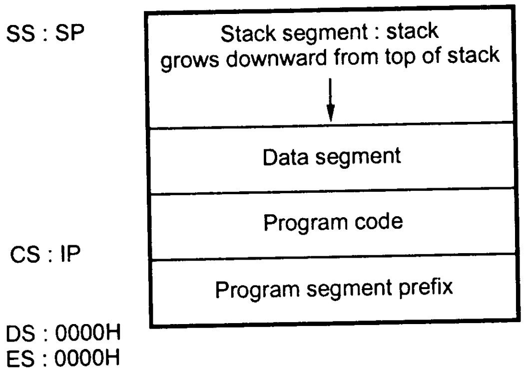


Fig. 4.4 A memory image of a typical .EXE program immediately after loading

After the above initialization, DOS discards the .EXE file header. With this initialization the CS and SS registers are set correctly, but your program has to set the DS and ES for its own data segment. This is illustrated in the sample .EXE program given below.

Sample .EXE Program

This source program gives the same result as the program listed as sample .COM program. However, it is EXE file rather than COM file. In this file two separate segments are defined : the data segment (named Data) and the code segment (named Code). The symbols Code and Data are arbitrary; any other symbols can be used.

```
PAGE     50,132
TITLE    WELCOME
; Writes WELCOME on the screen
.model small
.Data
    MES  DB        'WELCOME. $'
.Code
Start:   MOV AX,@Data       ; [ Initialize
         MOV DS,AX          ;   data segment ]
         MOV AX,09H
         MOV DX,OFFSET MES
         INT 21H            ; [ call DOS function
                            ;   to display message ]
         MOV AH,4CH
         INT 21H            ; [ call DOS function to
                            ;   terminate the program ]
         END Start
```

4.3.3 Comparison between .EXE and .COM Programs

Sr. No.	.COM Format	.EXE Format
1.	In COM program data, code, and stack reside in one segment.	.EXE program can have multiple code, data, and stack segment.
2.	The .COM files are compact and are loaded slightly faster than equivalent .EXE file, since these contain only the execution code.	.EXE files contain unique header, a relocation map, a checksum and other information used by DOS along with the execution code.
3.	Near subroutine are used in the .COM files.	.EXE programs can contain more than one code segment. So both near and far CALLs are used.
4.	CS starts at and IP at 0100H.	Not fixed.
5.	Maximum length of program (code and data) is 65,536 bytes (64 K) minus 256 bytes of PSP.	.EXE program can be as large as available memory.
6.	In .COM format, the size of the file is exact size of the program.	In .EXE formats, the size of the file is size of program plus the size of header.
7.	.COM program does not require file header.	.EXE programs need file header for relocation process.

Table 4.1 Comparison between .COM and .EXE files

4.3.4 Programmer's Template

The contents of the programmer's template must be entered for almost every program. If you create a file that contains the template, the overhead of writing template contents can be avoided. You can then create program directly from the template file by adding to it in appropriate place.

```
        ; This is a program template to which you can add
        ; program and data definitions.
        ; Insert program and data definitions where indicated
; model definition
        .model small
; stack definition.
        .stack 100
; data definitions
        .data
; code definitions
        .CODE
; Initialisation of data segment
START:    MOV AX,@DATA       ; [ Initialization
          MOV DS,AX          ;   of data segment ]
; User program
EXIT:     MOV AH,4CH         ; Terminate and
          INT 21H            ; Exit to DOS
          END START
```

4.3.5 Conversion of .ASM to .EXE and .EXE to .COM

To create .COM program, it is necessary to create .EXE program. To create .EXE program, .ASM program is first assembled using macro assembler to produce .OBJ (object) program. Then the object program is linked with the help of linker to produce .EXE program. Finally, .EXE program is converted into a .COM program with the help of EXE2BIN utility. To convert .EXE program to .COM program, program must meet the following prerequisites :

- It cannot contain more than one segment.
- It must be less than 64 kB in length.
- It must have an origin at 0100H.
- The first location in the file must be specified as the starting point in the source code's END directive.

Command formats :

MASM File_name.asm;
Link File_name.obj;
EXE2BIN sourcefile destinationfile

Example :

```
c:\masm61\bin\>MASM myprog.asm;
c:\masm61\bin\>LINK myprog.obj;
c:\masm61\bin\>EXE2BIN myprog.exe myprog.com
```

Note : Default extension for source file is .EXE, where as default extension for destination file is .BIN.

4.3.6 EXEC Function

The MS-DOS EXEC Function (INT 21H Function 4BH) allows to load .COM and .EXE programs from disk files, execute it, and then regain control when the program is finished. It also allows a program (called the parent) to load any other program (called the child) from disk, execute it and then regain control when the child program is finished. It builds a special data structure called a program segment prefix (PSP), in the transient program area (TPA). (Refer Fig. 4.5 on page 4-15). The PSP contains various linkages and pointers needed by the application programs. After building PSP, the EXEC function loads the program, just before the PSP and performs any relocation if necessary. It then sets up the segment registers and stack and transfers control to the program. So in all EXEC Function does following :

- Allocates memory for the new program.
- Builds the Program Segment Prefix () at the lowest area of memory.

- Loads the program above the PSP.
- Sets up appropriate registers, i.e. segment registers and the stack and transfers control to the new program.

For .EXE programs, the EXEC function may also do some additional processing like passing parameters from parent to child through the environment block.

The environment block holds certain information used by the system's command interpreter (usually COMMAND.COM) and may also hold information to be used by transient programs (.COM and .EXE programs).

4.3.7 Ending Program Execution

There are several ways to terminate current program execution and return control to MSDOS. These are explained below :

1. **INT 20H** : The INT 20H function ends execution of a .COM program, restores addresses for Ctrl + Break and critical errors, flushes register buffers and returns control to MSDOS. However, this function requires the address of the PSP in CS register.
2. **INT 21H : Function 00H** : This function terminates the current program, releases memory belongs to the program, flushes register buffers and returns control to MSDOS. However, this function also requires the address of the PSP in CS register.
3. **INT 21H : Function 31H** : This function terminates the execution of the currently executing program, passing a return code to the parent process, but reserves part or all of the program's memory so that it will not be overlaid by the next transient program to be loaded.
4. **INT 27H** : This function terminates the execution of the currently executing program but reverses part or all of its memory so that it will not be overlaid by the next transient program to be loaded. This function requires the address of the PSP in CS register and the offset of the last byte plus one (relative to the PSP) of program in the DX register.
5. **INT 21H : Function 4CH** : This function terminates the current program by passing a return code to the parent program. This function releases all memory belongs to the program, flushes register buffers and returns code in the AL register. The return code for normal completion of a program is usually 0 (zero). Because this function does not require the address of PSP in CS and it releases all memory belongs to the program, it is the standard, preferred method of program termination.

4.4 PSP (Structure Details)

As mention earlier, PSP contains various linkages and pointers needed by the application programs. It is a special data structure of 256 bytes. Fig. 4.5 shows the structure of program segment prefix. This structure is loaded by DOS before the transient program is loaded. It occupies the base of the memory block allocated to a transient program. Table 4.2 presents some of the important items in the PSP.

Offset	Contents
00H-01H	Contains a linkage to the MS-DOS process termination handler, which cleans up after the program has finished its job and performs a final exit.
02H-03H	Contains the address of the top of the transient program's allocated memory block. This information is used to determine whether it should request for extra memory to do its job or whether it has extra memory that it releases for use by other processes.
05-09H	Contains linkages to the MS-DOS function dispatcher, which performs disk operations, console input/output, and other such services at the request of the transient program.
0A-0DH	Contains the original contents of the interrupt vector (22H) for the termination.
0E-11H	Contains the original contents of Ctrl C (23H) interrupt vector.
12-15H	Contains the original contents of critical error handler (24H) interrupt vector.
2C-2DH	Contains the segment address of environment block.
5C-6BH	Default file control block #1
6C-7FH	Default file control block #2
80H	Length of the command tail not including return character at its end.
81H-FFH	The 128 byte area from 0080H to 00FFH contains command tail and is used as the default disk transfer area (DTA), which is set by MS-DOS before passing control to the transient program.

Table 4.2 Important areas in the PSP

Note :

1. (File Control Block) is a special data structure used to access a file.
2. Command tail is a remaining part of the command line that invoked the transient program, after the program's name.
3. DTA (Disk Transfer Area) : In file functions using FCB method data is always read to or written from the current disk transfer area (DTA), whose address is set with INT 21H function 1AH.

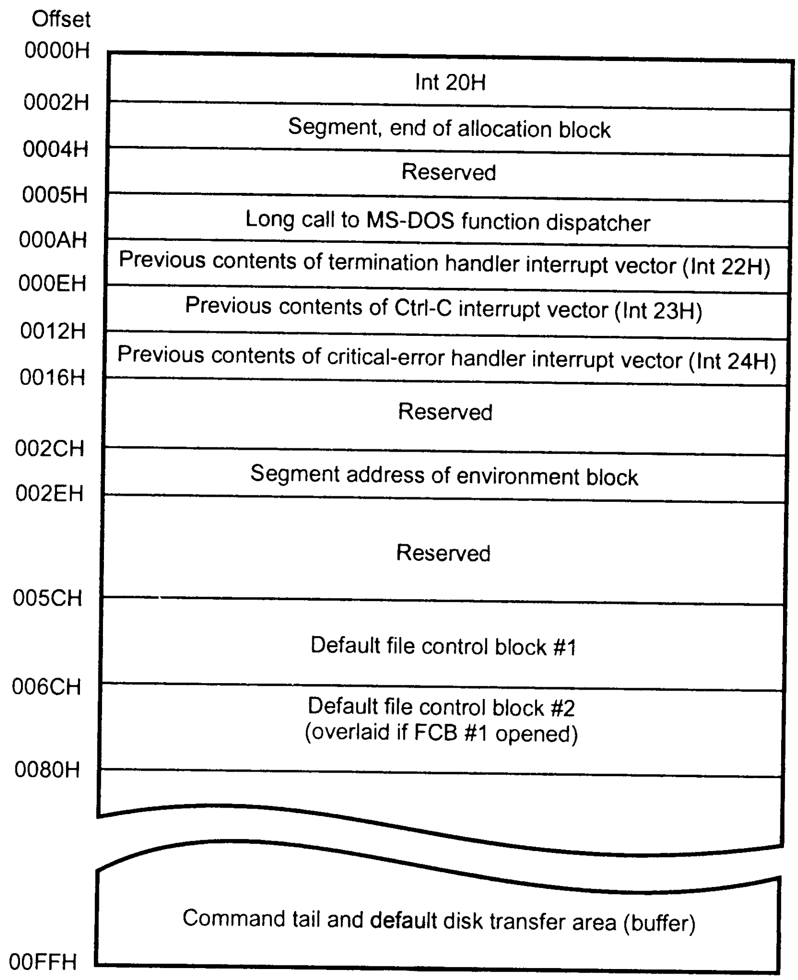


Fig. 4.5 Structure of program segment prefix

To summarize PSP does the following things :

1. Provides linkages to DOS required by the transient program.
2. Stores interrupt vector addresses of the parent program for the termination handler (INT 22H), the Ctrl C interrupt (INT 23H) and the critical error handler (INT 24H) needed by DOS for its own purpose.
3. Stores the command tail.
4. Provides two default file control blocks (FCBs).

4.5 DOS and BIOS Calls

4.5.1 Character Input Functions

Int 21H	Character input with echo	Function 01H

Reads a character from the standard input device and echoes it to the standard output device. If no character is ready, waits until one is available.

Calling parameters

AH = 01H

Returns

AL = 8-bit input data

Example : Read one character from the keyboard into register AL, echo it to the display, and store it in the variable char.

```
char    db   0              ; input character
        .
        .
        mov  ah,01h         ; function number
        int  21h            ; transfer to DOS
        mov  char,al        ; save character
        .
        .
```

Int 21H	Direct console I/O	Function 06H

Used by program that need to read and write all possible characters and control codes without any interference from the operating system.

Reads a character from the standard input device or writes a character to the standard output device. I/O may be redirected.

Calling parameter

```
AH = 06H
DL = function requested
     00H-FEH     if output request
     0FFH        if input request
```

Returns : Nothing, if called with DL = 00H-0FEH

```
If called with DL=FFH and a character is ready returns
Zero flag    =    clear
       AL    =    8-bit input data
If called with DL = FFH and no character is ready
Zero flag    =    set
```

Int 21H Unfiltered character input without echo Function 07H

Reads a character from the standard input device without echoing it to the standard output device. If no character is ready, waits until one is available.

Calling parameters

```
AH = 07H
```

Returns

```
AL = 8-bit input data
```

Example : Read a character from the standard input without echoing it to the display, and store it in the variable char.

```
char    db  0           ; input character
        .
        .
        mov ah,7        ; function number
        int 21h         ; transfer to MS-DOS
        mov char,al     ; save character
        .
        .
```

Int 21H Character input without echo Function 08H

Reads a character from the standard input device without echoing it to the standard output device. If no character is ready, waits until one is available.

Calling parameters

```
AH = 08H
```

Returns

```
AL = 8-bit input data
```

Example : Read a character from the standard input without echoing it to the display, allowing possible detection of Ctrl-C, and store the character in the variable char.

```
char    db  0
        .
        .
        mov ah,08h      ; function number
        int 21h         ; transfer to MS-DOS
        mov char,al     ; save character
```

Int 21H Buffered keyboard input Function 0AH (10)

Reads a string of bytes from the standard input device, up to and including an ASCII carriage return (0DH), and places them in an user-designated buffer. The characters are echoed to the standard output device.

Calling parameters

```
      AH = 0AH
   DS:DX = segment:offset of buffer
```

Returns : Nothing (data placed in buffer)

Notes :

The buffer used by this function has the following format :

Byte	**Contents**
0	maximum number of characters to read, set by program
1	number of characters actually read (excluding carriage return),set by MS - DOS
2	string read from keyboard or standard input, terminated by a carriage return (0DH)

If the buffer fills to one fewer than the maximum number of characters it can hold, subsequent input is ignored and the bell is sounded until a carriage return is detected.

Example : Read a string that is maximum of 80 characters long from the standard input device, placing it in the buffer named buffer.

```
buffer  db   81              ; maximum length of input
        db   0               ; actual length of input
        db   81 dup (0)      ; actual input placed here
        .
        .
        mov  ah,0ah          ; function number
        mov  dx,seg buffer;  input buffer address
        mov  ds,dx
        mov  dx,offset buffer
        int  21h             ; transfer to MS-DOS
        .
        .
```

Int 21H Check input status Function 0BH (11)

Checks whether a character is available from the standard input device.

Calling parameters

```
AH = 0BH
```

Returns

```
AL = 00H if no character is available
```

FFH if at least one character is available

Example : Test whether a character is available from the standard input.

```
.
.
mov  ah,0bh        ; function on number
int  21h           ; transfer to MS-DOS
or   al,al         ; character waiting?
```

```
jnz  avail        ; jump if char available
 .
 .
```

Int 21H **Flush input buffer and then input** **Function 0CH (12)**

Clears the standard input buffer and then invokes one of the character input functions. Input can be redirected.

Calling parameters

```
AH = 0CH
AL = number of input function to be invoked
     after resetting buffer (must be 01H, 06H,
     07H, 08H, or 0AH)
(if AL = 0AH)
 DS:DX = segment:offset of input buffer
```

Returns : (if called with AL = 01H, 06H, 07H, or 08H)

```
AL   =   8-bit input data
```

(if called with AL = 0AH)

Nothing (data placed in buffer)

4.5.2 Character Display Functions

Int 21H **Character output** **Function 02H**

Outputs the character to the standard output device.

Calling parameters

```
AH = 02H
DL = 8-bit data for output
```

Returns : Nothing

Example : Send the character "*" to the standard output device.

```
 .
 .
mov  ah,2         ; function number
mov  dl,'*'       ; character to output
int  21h          ; transfer to MS-DOS
 .
 .
```

Int 21H	Display string	Function 09H

Sends a string of characters to the standard output device. End of string is indicated by character $ (24H).

Calling parameters

```
AH = 09H
DS = segment:offset of string
```

Returns : Nothing

Example : Send the string, followed by a carriage return and line feed, to the standard output device.

```
cr   equ  0dh
lf   equ  0ah
msg  db   'MICROPROCESSOR',cr,lf,'$'
     .
     .
     mov  ah,09h         ; function number
     mov  dx,seg msg     ; address of string
     mov  ds,dx
     mov  dx,offset msg
     int  21h            ; transfer to MS-DOS
```

4.5.3 File Control Block Functions

Int 21H	Open file	Function 0FH (15)

Opens a file and makes it available for subsequent read/write operations.

Calling parameters

```
   AH = 0FH
DS:DX = segment:offset of file control block
```

Returns :

If function successful (file found)

AL = 00H

and FCB filled in by MS-DOS as follows :

drive field (offset 00H)	=	1 for drive A, 2 for drive B, etc.
current block field (offset 0CH)	=	00H
record size field (offset 0EH)	=	0080H
[2.0+] size field (offset 10H)	=	file size from directory
[2.0+] date field (offset 14H)	=	date stamp from directory
[2.0+] time field (offset 16H)	=	time stamp from directory

If function unsuccessful (file not found)

AL = FFH

Example : Attempt to open the file named TEST.DAT on the default disk drive.

```
myfcb   db   0                   ; drive = default
        db   'TEST'              ; filename, 8 characters
        db   'DAT'               ; extension, 3 characters
        db   25 dup (0)          ; remainder of FCB
         .
         .
        mov ah,0fh               ; function number
        mov dx,seg myfcb         ; address of FCB
        mov ds,dx
        mov dx,offset myfcb
        int 21h                  ; transfer to MS-DOS
        or al,al                 ; check status
        jnz error                ; jump if open failed
         .
         .
```

Int 21H Close file Function 10H (16)

Closes a file, flushes all MS-DOS internal disk buffers associated with the file to disk, and updates the disk directory if the file has been modified or extended.

Calling parameter

```
AH    = 10H
DS:DX = segment:offset of file control block
```

Returns

If function successful (directory update successful)

```
AL  =  00H
```

If function unsuccessful (file not found in directory)

```
AL  =  FFH
```

Example : Close the file that was previously opened using the file control block named myfcb.

```
myfcb   db   0                   ; drive = default
        db   'TEST'              ; filename, 8 characters
        db   'DAT'               ; extension, 3 characters
        db   25 dup (0)          ; remainder of FCB
         .
         .
        mov ah, 10h              ; function number
        mov dx, seg myfcb        ; address of FCB
        mov ds, dx
        mov dx, offset myfcb
        int 21h                  ; transfer to MS-DOS
        or al, al                ; check status
        jnz error                ; jump if close failed
         .
         .
```

Int 21H — Delete file — Function 13H (19)

Deletes all matching files from the current directory on the default or specified disk drive.

Calling parameter

```
    AH = 13H
 DS:DX = segment:offset of file control block
```

Returns :

If function successful (file or files deleted)

```
    AL  =   00H
```

If function unsuccessful (no matching files were found, or at least one matching file was read-only)

```
    AL  =   FFH
```

Example :

Delete the file TEST.DAT from the current disk drive and directory.

```
myfcb   db  0                    ; drive = default
        db  'TEST'               ; filename, 8 characters
        db  'DAT'                ; extension, 3 characters
        db  25 dup (0)           ; remainder of FCB
         .
         .
        mov ah,13h               ; function number
        mov dx,seg myfcb         ; address of FCB
        mov ds,dx
        mov dx,offset myfcb
        int 21h                  ; transfer to MS-DOS
        or al,al                 ; check status
        jnz error                ; jump if close failed
         .
         .
```

Int 21H — Sequential read — Function 14H (20)

Reads the next sequential block of data from a file, then increments the file pointer appropriately.

Calling parameter

```
    AH = 14H
 DS:DX = segment:offset of previously opened file
         control block
```

Returns

```
       AL = 00H  if read successful
```

```
01H  if end of file
02H  if segment wrap
03H  if partial record read at end of file
```

Example : Read 512 bytes of data from the file specified by the previously opened file control block myfcb.

```
myfcb    db   0                       ; drive = default
         db   'TEST'                  ; filename, 8
                                      ; characters
         db   'DAT'                   ; extension, 3
                                      ; characters
         db   25 dup (0)              ; remainder of FCB
           .
           .
         mov ah,14h                   ; function number
         mov dx,seg myfcb             ; address of FCB
         mov ds,dx
         mov dx,offset myfcb          ; set record size
         mov word ptr myfcb+0eh,512
         int 21h                      ; transfer to MS-DOS
         or  al,al                    ; check status
         jnz error                    ; jump if read failed
              .
              .
```

Int 21H	Sequential write	Function 15H (21)

Writes the next sequential block of data from a file, then increments the file pointer appropriately.

Calling parameter

```
      AH = 15H
   DS:DX = segment:offset of previously opened file
           control block
```

Returns

```
   AL  =   00H     if write successful
           01H     if disk is full
           02H     if segment wrap
```

Example : Write 512 bytes of data to the file specified by the previously opened file control block myfcb.

```
myfcb    db   0              ; drive = default
         db   'TEST'         ; filename, 8 characters
         db   'DAT'          ; extension, 3 characters
         db   25 dup (0)     ; remainder of FCB
          .
          .
         mov ah,15h           ; function number
         mov dx,seg myfcb     ; address of FCB
```

```
            mov ds,dx
            mov dx,offset myfcb   ; set record size
            mov word ptr myfcb+0eh, 1024
            int 21h               ; transfer to MS-DOS
            or  al, al            ; check status
            jnz error             ; jump if write failed
              .
              .
```

Int 21H Create file Function 16H (22)

Creates a new directory entry in the current directory or truncates any existing file with the same name to zero length. Opens the file for subsequent read/write operations.

Calling parameter

```
     AH = 16H
  DS:DX = segment:offset of previously opened file
           control block
```

Returns : If function successful (file was created or truncated)

```
AL = 00H
```

and FCB filled in by MS-DOS as follows :

drive field (offset 00H) = 1 for drive A, 2 for drive B, etc.
current block field (offset 0CH) = 00H
record size field (offset 0EH) = 0080H
[2.0+]size field (offset 10H) = file size from directory
[2.0+]date field (offset 14H) = date stamp from directory
[2.0+] time field (offset 16H) = time stamp from directory
If function unsuccessful (directory full)

```
AL = FFH
```

Example : Create a file in the current directory using the name in the file control block myfcb.

```
myfcb       db  0            ; drive = default
            db  'TEST'       ; filename, 8 characters
            db  'DAT'        ; extension, 3 characters
            db  25 dup (0)   ; remainder of FCB
              .
              .
            mov ah,16h            ; function number
            mov dx,seg myfcb      ; address of FCB
            mov ds,dx
            mov dx,offset myfcb
            int 21h               ; transfer to MS-DOS
            or  al, al            ; check status
            jnz error             ; jump if create failed
              .
              .
```

Int 21H Rename file Function 17H (23)

Alters the name of all matching files in the current directory on the disk in the specified drive.

Calling parameter

```
AH = 17H
DS:DX = segment:offset of "special" file control
        block
```

Returns : If function successful (one or more files are renamed)

```
AL = 00H
```

If function unsuccessful (no matching files, or new filename matched an existing file)

```
AL = FFH
```

Example : Rename the file OLDNAME.DAT to NEWNAME.DAT.

```
myfcb   db   0              ; drive = default
        db   'OLDNAME'      ; old file name, 8 characters
        db   'DAT'          ; old extension, 3 characters
        db   6 dup (0)      ; reserved area
        db   'NEWNAME'      ; new file name, 8 characters
        db   'DAT'          ; new extension, 3 characters
        db   14 dup (0)     ; reserved area
          .
          .
        mov ah,17h               ; function number
        mov dx,seg myfcb         ; address of FCB
        mov ds,dx
        mov dx,offset myfcb
        int 21h                  ; transfer to MS-DOS
        or al,al                 ; check status
        jnz error                ; jump if close failed
          .
          .
```

Int 21H Get file size Function 23H (35)

Searches for a matching file in the current directory; if one is found, updates the FCB with the file's size in terms of number of records.

Calling parameters :

```
AH = 23H
DS:DX = segment: offset of unopened file control
        block
```

Returns : If function successful (matching file found)

```
AL = 00H
```

and FCB relative-record field (offset 21H) set to the number of records in the file.

If function unsuccessful (no matching file found)

```
AL  =  FFH
```

Example : Determine the size in bytes of the file MICRO.DAT

```
myfcb   db   0                  ; drive - default
        db   'MICRO'            ; filename, 8 chars
        db   'DAT'              ; extension, 3 chars
        db   25 dup (0)         ; remainder of FCB
        .
        .
        .
        mov  ah,23h        ; function number
        mov  dx,seg myfcb  ; address of FCB
        mov  ds,dx
        mov  dx,offset myfcb
                           ; record size-1 byte
        mov  word ptr myfcb+0eh, 1
        int  21h           ; transfer to MS-DOS
        or   al,al         ; check status
        jnz   error        ; jump if no file
                           ; get file size in bytes
        mov   ax, word ptr myfcb+21h
        mov   dx, word ptr myfcb+23h
        .
        .
        .
```

4.5.4 Handle Functions

Int 21H	Create file	Function 3CH (60)

Creates a new file in the designated or default directory on the designated or default disk drive. If the specified file already exists, it is truncated to zero length. In either case, the file is opened and a handle is returned that can be used by the program for subsequent access to the file.

Calling parameters

```
AH = 3CH
CX = file attribute (bits may be combined)
      Bit(s)   Significance (if set)
      0        read-only
      1        hidden
      2        system
      3        volume label
      4        reserved (0)
      5        archive
      6-15 reserved (0)
   DS:DX = segment:offset of ASCII path name
```

Returns : If function successful

```
Carry flag = clear
        AX = handle
```

```
If function failed
Carry flag = set
      AX = error code
```

Example : Create and open,or truncate to zero length and open, the file

```
C:\MBS\PRO1.ASM and save the handle for subsequent
  access to the file.
fname         db       'C:\MBS\PRO1.ASM',0
fhandle       dw       ?
   .
   .
   mov  ah,3ch              ;    function number
   xor  cx,cx               ;    normal attribute
   mov  dx,seg fname        ;    address of path name
   mov  ds,dx
   mov  dx,offset fname
   int  21h                 ;    transfer to MS-DOS
   jc   error               ;    jump if create failed
   mov  fhandle,ax          ;    save file handle
   .
   .
```

Int 21H — Open file — Function 3DH (61)

Opens the specified file in the designated or default directory on the designated or default disk drive. A handle is returned which can be used by the program for subsequent access to the file.

Calling parameters

```
AH = 3DH
AL = access mode
        Bit(s)   Significance
        0-2          access mode
                     000 = read access
                     001 = write access
                     010 = read/write access
        3        reserved (0)
        4-6      sharing mode (MS-DOS versions 3.0
                 and later)
                     000 = compatibility mode
                     001 = deny all
                     010 = deny write
                     011 = deny read
                     100 = deny none
        7        inheritance flag (MS-DOS versions 3.0
                 & later)
                     0 = child process inherits handle
                     1 = child does not inherit handle
DS:DX = segment:offset of ASCII path name
```

Returns : If function successful

```
Carry flag = clear
        AX = handle
```

If function unsuccessful

```
Carry flag = set
        AX = error code
```

Example : Open the file C:\\PRO1.ASM for both reading and writing, and save the handle for subsequent access to the file.

```
fname        db   'C:\MBS\PRO1.ASM',0
fhandle      dw   ?
                 .
                 .
                 mov  ah,3dh        ;  function number
                 mov  al,02h        ;  mode - read/write
                 mov  dx,seg fname  ;  address of path name
                 mov  ds,dx
                 mov  dx,offset fname
                 int  21H           ;  transfer to MS-DOS
                 jc   error         ;  jump if open failed
                 mov  fhandle,ax    ;  save file handle
                 .
                 .
```

Int 21H — Close file — Function 3EH (62)

Given a handle that was obtained by a previous successful open or create operation, flushes all internal buffers associated with the file to disk, closes the file, and releases the handle for reuse. If the file was modified, the time and date stamp and file size are updated in the file's directory entry.

Calling parameters

```
AH = 3EH
BX = handle
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
        AX = error code
```

Example : Close the file whose handle is saved in the variable fhandle.

```
fhandle  dw   0
             .
             .
             mov  ah,3eh        ;  function number
             mov  bx,fhandle    ;  file handle
```

```
                int  21h            ;  transfer to MS-DOS
                jc   error          ;  jump if close failed
                .
                .
```

Int 21H — Read file or device — Function 3FH (63)

Given a valid file handle from a previous open or create operation, a buffer address, and a length in bytes, transfers data at the current file-pointer position from the file into the buffer and then updates the file pointer position.

Call parameters :

```
        AH =  3FH
        BX =  handle
        CX =  number of bytes to read
     DS:DX =  segment:offset of buffer
```

Returns : If function successful

```
   Carry flag =  clear
           AX =  bytes transferred
```

If function unsuccessful

```
   Carry flag =  set
           AX =  error code
```

Example : Using the file handle from a previous open or create operation, read 512 bytes at the current file pointer into the buffer named buff.

```
buff      db   512 dup (?)   ; buffer for read
fhandle   dw   ?             ; contains file handle
          .
          .
          .
          mov ah,3fh              ; function number
          mov dx, seg buff        ; buffer address
          mov ds, dx
          mov dx, offset buff
          mov bx, fhandle         ; file handle
          mov cx, 512             ; length to read
          int 21h                 ; transfer to MS-DOS
          jc  error               ; jump, read failed
          cmp ax, cx              ; check length of read
          jl  done                ; jump, end of file
          .
          .
```

Int 21H Write file or device Function 40H (64)

Given a valid file handle from a previous open or create operation, a buffer address, and a length in bytes, transfers data from the buffer into the file and then updates the file pointer position.

Call parameters

```
          AH = 40H
          BX = handle
          CX = number of bytes to write
       DS:DX = segment:offset of buffer
```

Returns : If function successful

```
    Carry flag = clear
            AX = bytes transferred
```

If function unsuccessful

```
    Carry flag = set
            AX = error code
```

Example : Using the handle from a previous open or create operation, write 512 bytes to disk at the current file pointer from the buffer named buff.

```
buff      db   512 dup (?)        ; buffer for write
fhandle   dw   ?                  ; contains file handle
          .
          .
          mov ah,40h              ; function number
          mov dx, seg buff        ; buffer address
          mov ds, dx
          mov dx, offset buff
          mov bx, fhandle         ; file handle
          mov cx, 512             ; length to write
          int 21h                 ; transfer to MS-DOS
          jc  error               ; jump, write failed
          cmp ax, 512             ; entire record written?
          jne error               ; no, jump
```

Int 21H Delete file Function 41H (65)

Deletes a file from the specified or default disk and directory.

Calling parameters

```
          AH = 41H
       DS:DX = segment:offset of ASCIIZ pathname
```

Returns : If function successful

```
    Carry flag = clear
```

If function unsuccessful

```
    Carry flag = set
            AX = error code
```

Example : Delete the file named MICRO.DAT from the directory \MYDIR on drive C.

```
fname    db   'C:\MYDIR\MICRO.DAT',0
          .
          .
mov ah,41h              ; function number
mov dx,seg fname        ; filename address
mov ds,dx
mov dx,offset fname
int 21h                 ; transfer to MS-DOS
jc  error               ; jump if delete failed
.
.
.
```

INT 21H	Move file pointer	Function 42H (66)

DOS maintains a file pointer. The open file operation initialize file pointer to 0 and subsequent sequential reads and writes increment file pointer by record.

Call with :

```
AH = 42H
AL = method code
     00H absolute offset from start of file
     01H signed offset from current file pointer
     02H signed offset from end of file
BX = handle
CX = most significant half of offset
DX = least signficant half of offset
```

Returns : If function successful

```
Carry flag = clear
        DX = most significant half of resulting file
             pointer
        AX = least significant half of resulting file
             pointer
```

If function unseccessful

```
Carry flag = set
        AX = error code
```

Int 21H	Rename file	Function 56H (86)

Renames a file and/or moves its directory entry to a different directory on the same disk. In MS-DOS version 3.0 and later, this function can also be used to rename directories.

Calling parameter

```
   AH = 56H
  DS: = segment:offset of current ASCIIZ pathname
ES:DI = segment:offset of new  pathname
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
        AX = error code
```

Example : Change the name of the file MYFILE.DAT in the directory \MYDIR on drive C to MYTEXT.DAT. At the same time, move the file to the directory \SYSTEM on the same drive.

```
oldname   db   'C:\MYDIR\MYFILE.DAT',0    ; drive = default
newname   db   'C:\SYSTEM\MYTEXT.DAT',0
          .
          .
          mov ah, 56h               ; function number
          mov dx, seg oldname       ; old filename address
          mov ds, dx
          mov dx, offset oldname
          mov di, seg newname       ; new filename address
          mov es, di
          mov di, offset newname
              int 21h               ; transfer to MS-DOS
              jc error              ; jump if rename
                                    ; failed
               .
               .
```

4.5.5 Memory Management Functions

Int 21H	**Allocate memory block**	**Function 48H (72)**

Allocates a block of memory and returns a pointer to the beginning of the allocated area.

Calling parameter

```
AH = 48H
BX = number of paragraphs of memory needed
```

Returns : If function successful

```
Carry flag = clear
        AX = base segment address of allocated block
```

If function unsuccessful

```
Carry flag = set
        AX = error code
           = size of largest available block
             (paragraphs)
```

Example : Request a 64 kB block of memory for use as a buffer.

```
bufseg   dw   ?               ; segment base of new block
          .
          .
```

```
        mov  ah,48h        ; function number
        mov  bx,1000h      ; block size (paragraphs)
        int  21h           ; transfer to MS-DOS
        jc   error         ; jump if allocation failed
        mov  bufseg,ax     ; save segment of new block
```

Int 21H Release memory block Function 49H (73)

Releases a memory block and makes it available for use by other programs.

Calling parameter

```
AH = 49H
ES = segment of block to be released
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
        AX = error code
```

Example : Release the memory block that was previously allocated in the example for 21H Function 48H.

```
bufseg  dw   ?              ; segment base of block
        .
        .
        mov  ah,49h         ; function number
        mov  es,bufseg      ; base segment of block
        int  21h            ; transfer to MS-DOS
        jc   error          ; jump if release failed
        .
        .
```

Int 21H Resize memory block Function 4AH (74)

Dynamically shrinks or extends a memory block, according to the needs of an application program.

Calling parameter

```
AH = 4AH
   = desired new block size in paragraphs
ES = segment of block to be modified
```

Returns : If function successful

```
Carry flag = clear
```

If function unsuccessful

```
Carry flag = set
        AX = error code
           = maximum block size
             available (paragraphs)
```

Example : Resize the memory block that was allocated in the example for Int 21H Function 48H, shrinking it to 32 kB.

```
bufseg  dw  ?            ; segment base of block
        .
        .
        mov ah,4ah       ; function number
        mov bx,0800h     ; new size (paragraphs)
        mov es,bufseg    ; segment base of block
        int 21h          ; transfer to MS-DOS
        jc  error        ; jump, resize failed
        mov bufseg, ax   ; save segment of new block
        .
```

Int 15H	Move extended memory block	Function 87H (135)

Transfers data between conventional memory and extended memory.

Calling parameter

```
   AH = 87h
   CX = number of words to move
ES:SI = segment:offset of Global Descriptor Table
```

Returns : If function successful

```
Carry flag = clear
        AH = 00H
```

If function unsuccessful

```
Carry flag = set
        AH = status
        01H if RAM parity error
        02H if exception interrupt error
        03H if gate address line 20 failed
```

Int 15H	Get extended memory size	Function 88H(136)

Returns the amount of extended memory installed in the system

Calling parameter

```
AH = 88H
```

Returns :

```
AX = amount of extended memory (in kB)
```

4.5.6 Display Functions Provided by ROM BIOS

Int 10H	Set video mode	Function 00H

Selects the current video display mode. Also selects the active video controller, if more than one video controller is present.

Calling parameters

```
AH = 00H
AL = video modes
```

Returns : Nothing

Different video modes

Mode	Resolution	Colors	Text/graphics
00H	40-by-25 color burst off	16	text
01H	40-by-25	16	text
02H	80-by-25 color burst off	16	text
03H	80-by-25	16	text
04H	320-by-200	4	graphics
05H	320-by-200 color burst off	4	graphics
06H	640-by-200	2	graphics
07H	80-by-25	2[1]	text
08H	160-by-200	16	graphics
09H	320-by-200	16	graphics
0AH	640-by-200	4	graphics
0BH	reserved		
0CH	reserved		
0DH	320-by-200	16	graphics
0EH	640-by-200	16	graphics
0FH	640-by-350	2[2]	graphics
10H	640-by-350	4	graphics
10H	640-by-350	16	graphics
11H	640-by-480	2	graphics
12H	640-by-480	16	graphics
13H	320-by-200	256	graphics

Int 10H Set cursor type Function 01H

Selects the starting and ending lines for the blinking hardware cursor in text display modes.

Calling parameters

```
            AH = 01H
CH bits 0-4 = starting line for cursor
CL bits 0-4 = ending line for cursor
```

Note : Cursor can be disabled by setting CH = 20H

Returns : Nothing

Int 10H Set cursor position Function 02H

Positions the cursor on the display, using text co-ordinates.

Calling parameters

```
AH = 02H
BH = page
DH = row (y co-ordinate)
DL = column (x co-ordinate)
```

Returns : Nothing

Int 10H Get cursor position Function 03H

Obtains the current position of the cursor on the display, in text co-ordinates.

Calling parameters

```
AH = 03H
BH = page
```

Returns :

```
CH = starting line for cursor
CL = ending line for cursor
DH = row (y co-ordinate)
DL = column (x co-ordinate)
```

Int 10H Read character and attribute at cursor Function 08H

Writes an ASCII character and its attribute to the display at the current cursor position.

Calling parameters

```
AH = 08h
AL = character
```

```
BH = page
BL = attribute (text modes) or color
     (graphics modes)
CX = count of characters to write
     (replication factor)
```

Returns : Nothing

Int 10H **Write character at cursor** **Function 0AH (10)**

Writes an ASCII character to the display at the current cursor position. The character receives the attribute of the previous character displayed at the same position.

Calling parameters

```
AH = 0AH
AL = character
BH = page
BL = color
CX = count of characters to write
     (replication factor)
```

Returns : Nothing

Int 10H **Write graphics pixel** **Function 0CH (12)**

Draws a point on the display at the specified graphics co-ordinates.

Calling parameters

```
AH = 0CH
AL = pixel value
BH = page
CX = column (graphics x co-ordinate)
DX = row (graphics y co-ordinate)
```

Returns : Nothing

Int 10H **Read graphics pixel** **Function 0DH (13)**

Obtains the current value of the pixel on the display at the specified graphics co-ordinates.

Calling parameters

```
AH = 0DH
BH = page
CX = column (graphics x co-ordinate)
DX = row (graphics y co-ordinate)
```

Returns : Nothing

```
AL = pixel value
```

4.5.7 Printer Functions

Int 21H	Printer output	Function 05H

Sends a character to the standard list device. The default device is the printer on the first parallel port (LPT1)

Calling parameters

```
AH = 05H
DL = 8-bit data for output
```

Returns : Nothing

Example : Output character "*" to the list device.

```
.
.
mov  ah,5          ; function number
mov  dl,'*'        ; character to output
int  21h           ; transfer to MS-DOS
```

Int 17H	Write character to printer	Function 00H

Sends a character to the specified parallel printer interface port and returns the current status of the port.

Calling parameters

```
AH = 00H
AL = character
DX = printer number (0 = LPT1, 1 = LPT2,
2  = LPT3)
```

Returns :

```
AH = status
Bit  Significance (if set)
0        printer timed-out
1        unused
2        unused
3        I/O error
4        printer selected
5        out of paper
6        printer acknowledge
7        printer not busy
```

Int 17H	Initialize printer port	Function 01H

Initializes the specified parallel printer interface port and returns its status.

Calling parameters

```
AH = 01H
```

DX = printer number (0 = LPT1, 1 = LPT2, 2 = LPT3)

Returns :

AH = status (see Int 17H Function 00H)

Int 17H	Get printer status	Function 02H

Returns the current status of the specified parallel printer interface port.

Calling parameters

AH = 02H
DX = printer number (0 = LPT1, 1 = LPT2, 2 = LPT3)

Returns : AH = status (see Int 17H Function 00H)

Review Questions

1. *Describe the important functions of operating system.*
2. *What are the important components of DOS ?*
3. *How DOS is loaded ?*
4. *Explain the structure of .COM and .EXE programs.*
5. *Compare .COM and .EXE programs.*
6. *Explain the procedure to generate .COM and .EXE files from .ASM files.*
7. *What are the functions of EXEC function ?*
8. *Explain various method of program termination.*
9. *What is PSP ? What are its functions ?*
10. *Draw and explain the structure of PSP.*
11. *What is BIOS ?*
12. *Explain the difference between BIOS and DOS.*

❑❑❑

5

Assembly Language Programs

In this chapter, we see the programs involving logical, branch and call instructions, sorting, evaluation of arithmetic expressions and string manipulation. Most of the programs use DOS function calls. The details of DOS function calls are given in chapter 4.

Program 7 : Read keyboard input and display it on monitor

```
TITLE Read Keyboard Input and Display it on Monitor
.model small
.code
start:    mov ax,@data                   ; [loads the address of data
              mov ds,ax                  ; segment in DS]
back:     mov ah,01
              int 21h
              cmp al,'0'
              jz Last
              jmp back
Last:     mov ah,4ch                     ; [ Exit
              int 21h                    ; to DOS ]
end start
end
```

Program 8 : Addition of two 32-bit numbers

```
; This program adds two numbers
TITLE Addition of two 32-bit numbers
.model small
.data
no1       dd   8111FFFFh
no2       dd    92224444h
result    dd    ?
carry     db    0
.code
start:    mov ax, @data              ; [loads the address of data
          mov ds, ax                 ;  segment in DS]
          mov ax,word ptr no1        ; Get the LS word of first
                                     ; number in AX add ax,word
                                     ; ptr no2 Add the LS word of
                                     ; second number to it
```

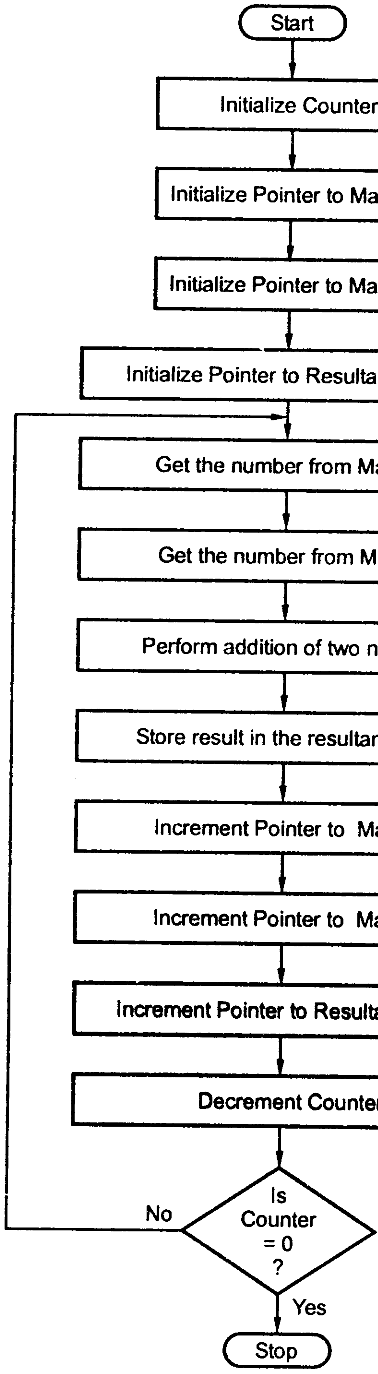
```
        mov word ptr result,ax; Save LS word of result
        mov bx, offset[no1]
        mov ax,word ptr [bx+2]; Get the MS word of first
                              ; number in AX
            mov bx, offset[no2]
            adc ax,word ptr [bx+2]    ; Add the MS word of second
                                      ; number to it with carry
            mov bx, offset result
            mov [bx+2],ax             ; Save MS word of result
            adc carry,0               ; save any carry after
                                      ; MS word addition
            mov ah,4ch                ; [ Exit
            int 21h                   ; to DOS ]
            end start
            end
```

Program 9 : Addition of 3 × 3 matrix

```
; This program adds 3 × 3 matrix. The matrices are stored in
; form of lists (row wise).
TITLE Addition of 3 × 3 Matrix
.model small
.data
m1      db 10h,20h,30h,40h,50h,60h,70h,80h,90h
m2      db 10h,20h,30h,40h,50h,60h,70h,80h,90h
result dw 9 dup(0)
.code
start:      mov ax,@data          ; [loads the address of data
            mov ds,ax             ; segment in DS]
            mov cx,9              ; Initialise the counter
            mov di, offset m1     ; Initialise the pointer to
                                  ; matrix1
            mov bx, offset m2     ; Initialise the pointer to
                                  ; matrix2
            mov si, offset result ; Initialise the pointer to
                                  ; resultant matrix
back:       mov ah,00             ; Make MSB of result zero
            mov al,[di]           ; Get the number from matrix1
            add al,[bx]           ; Get the number from matrix2
                                  ; and add it in corresponding
                                  ; number of matrix1
            adc ah,00             ; Save the carry of addition
                                  ; in MSB
            mov [si],ax           ; Store the result in
                                  ; corresponding position of
                                  ; resultant matrix
            inc di                ; increment pointer to matrix1
            inc bx                ; increment pointer to matrix2
            inc si                ; [ increment pointer
            inc si                ; to resultant matrix ]
            loop back             ; Repeat the process for all
                                  ; matrix elements
```

```
                mov  ah,4ch              ; [ Exit
                int  21h                 ;   to DOS ]
end  start
end
```

Flowchart :



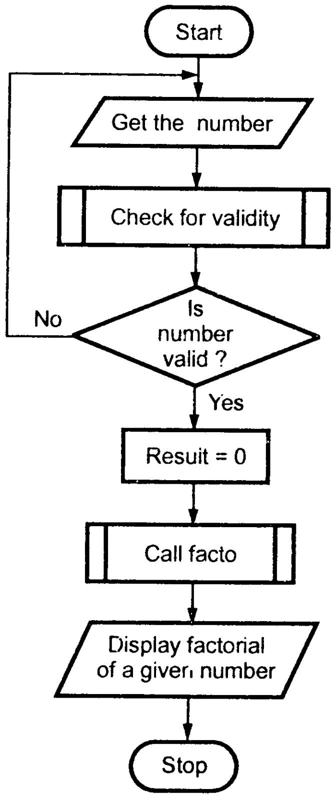
Program 10 : Program to read a password and validate user

```
.MODEL SMALL
.DATA
.STACK 100
PASS DB 'MBS1234'
MES1 DB 10,13,'ENTER 7 CHARACTER PASSWORD $'
MES2 DB 10,13,'PASSWORD IS CORRECT $'
MES3 DB 10,13,'INVALID PASSWORD$'
.CODE
START:    MOV AX,@DATA          ;[ Initialise
          MOV DS,AX             ; data segment ]
          MOV AH,09H
          LEA DX,MES1
          INT 21H               ; Display message
          MOV CL,00             ; Clear count
          MOV DH,00H            ; Clear number of match
          XOR DI,DI             ; Initialise pointer
          .WHILE CL != 7        ; Check if count = 7 if not
                                ; Continue
          MOV AH,07H
          INT 21H               ; Read character
          PUSH AX               ; Save character
          MOV AH,02H            ; [ Display
          MOV DL, '*'           ; '*' instead of
          INT 21H               ; character ]
          POP AX                ; Restore character
          LEA BX,PASS           ; [ Set pointer
          MOV AH,[BX+DI]        ; to password ]
          .IF AL==AH            ; Compare read character with
                                ; password
          ADD DH,01             ; Increament match count if match
                                ; occurs
          .ENDIF
          INC DI                ; Increment pointer
          INC CL                ; Increment counter
          .ENDW
          .IF DH == 7           ; [ if match count = 7
          MOV AH,09H            ; display message
          LEA DX,MES2           ; password is correct ]
          INT 21H
          .ELSE                 ; [ if match count <> 7
          MOV AH,09H            ; display message
          LEA DX,MES3           ; password is wrong ]
          INT 21H
          .ENDIF
          MOV AH,4CH            ; [ Exit to
          INT 21H               ; DOS ]
END START
END
```

Program 11 : Program to calculate factorial of a number

(Softcopy of this program, P18.asm is available at www.vtubooks.com)

Flowchart :




```
.MODEL SMALL
.STACK 100
.DATA
        MS1  DB 10,13,'ENTER THE NO.:$'
        MS2  DB 10,13,'THE FACTORIAL IS : $'
        NUM      DW 0
        ANS  DW 0
.CODE
START:       MOV     DX,@data       ; [ Initialise
             MOV     DS,DX          ; data segment ]
ERROR:       LEA     DX,MS1
             MOV     AH,09H         ; Display message MS1
             INT     21H
             MOV     AH,01H         ; Input number with echo
             INT     21H
             CMP     AL,30H         ; If zero display 1
             JE      DISPLY2
             CMP     AL,30H         ; If < 30 then input
```

```
          JB      ERROR                ; Next no
          CMP     AL,39H               ; If >39 then input
          JA      ERROR                ; Next no
          SUB     AL,30H               ; Convert to HEX
          MOV     AH,00H
          SUB     SP,0004H             ; Space in stack for
          PUSH    AX                   ; Factorial
          CALL    FACTO
          ADD     SP,0002              ; After execution
          POP     AX                   ; Of facto space for
          POP     DX                   ; Result
          MOV     BX,0010              ; Convert HEX to BCD
          MOV     CX,0006              ; Max input no is 9
BACK:     DIV     BX                   ; To get remainder
          OR      DX,0030H             ; Convert to ASCII
          PUSH    DX
          XOR     DX,DX                ; Clear DX
          LOOP    BACK
          LEA     DX,MS2               ; Output MS2
          MOV     AH,09
          INT     21H
          MOV     CX,0006
DISPLY1:  POP     DX
          MOV     AH,02H               ; Output factorial
          INT     21H
          LOOP    DISPLY1
          JMP     LAST
DISPLY2:  MOV     AH,09
          LEA     DX,MS2               ; Display factorial of
          INT     21H                  ; Zero = 1
          MOV     AH,02H
          MOV     DL,31H
          INT     21H
LAST:     MOV     AH,4CH               ; [ Terminate and
          INT     21H                  ;   Exit to DOS ]
          FACTO   PROC
          PUSHF
          PUSH    AX
          PUSH    DX
          PUSH    BP
          MOV     BP,SP                ; Point BP at TOS
          MOV     AX,[BP + 10]         ; Copy no from stack to
          CMP     AX,0001H             ; AX & if no not = 1 then
                                       ; GO_ON
          JNE     GO_ON                ; To compute factorial
          MOV     WORD PTR[BP+12],0001H
                                       ; Else load FFACT
          MOV     WORD PTR [BP+14],0000H
                                       ; 0 and 1 in stack
          JMP     EXIT
```

```
GO_ON:          SUB     SP,0004H            ; Space for preliminary
                DEC     AX                  ; Factorial
                PUSH    AX
                CALL    FACTO
                MOV     BP,SP
                MOV     AX,[BP+2]           ; Last (N - 1)! from
                                            ; stack to AX
                MUL     WORD PTR [BP+16]    ; Multiply by previous N
                MOV     [BP+18],AX          ; Copy new facto to stack
                MOV     [BP+20],DX
                ADD     SP,0006H            ; Point SP at pushed REGR
EXIT:           POP     BP
                POP     DX
                POP     AX
                POPF
                RET
FACTO           ENDP
                END     START
```

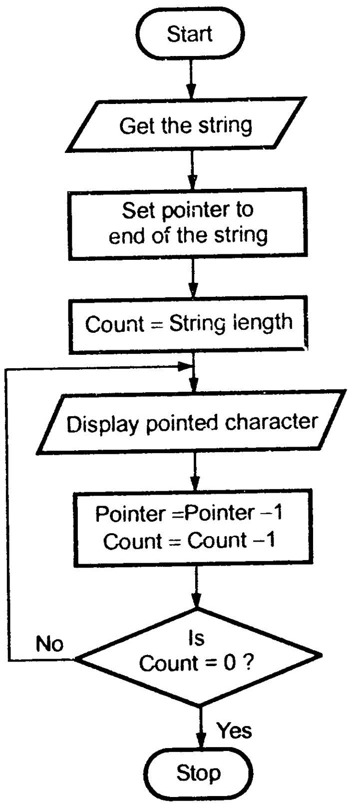
Program 12 : Reverse the words in string

(Softcopy of this program, P19.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.STACK 100
.DATA
TITLE   REVERSE THE WORDS IN STRING
        M1        DB 10,13, 'ENTER THE STRING:$'
        M2        DB 10,13, 'THE REVERSE STRING :$'
        BUFF      DB 80
                  DB 0
                  DB 80 DUP(0)
    COUNTER1      DW 0
    COUNTER2      DW 0
.CODE
START:            MOV     AX,@data        ; [ Initialise
                  MOV     DS,AX           ;   data segment ]
                  MOV     AH,09H          ; Display message M1.
                  MOV     DX,OFFSET M1
                  INT     21H
                  MOV     AH,0AH
                  LEA     DX,BUFF         ; I/P the string.
                  INT     21H
                  MOV     AH,09H
                  MOV     DX,OFFSET M2    ; Display message M2
                  INT     21H
                  LEA     BX,BUFF
                  INC     BX
                  MOV     CH,00H          ; [ Take character
                  MOV     CL,BUFF + 1     ;   count in
                  MOV     DI,CX           ;   DI ]
```

```
BACK:       MOV     DL,[BX+DI]      ; Point to the end
                                    ; character and read it
            MOV     AH,02H
            INT     21H             ; Display the character
            DEC     DI              ; Decrement count
            JNZ     BACK            ; Repeat until count is 0
EXIT:       MOV     AH,4CH          ; [ Terminate
            INT     21H             ;    Exit to DOS ]
            END     START
```

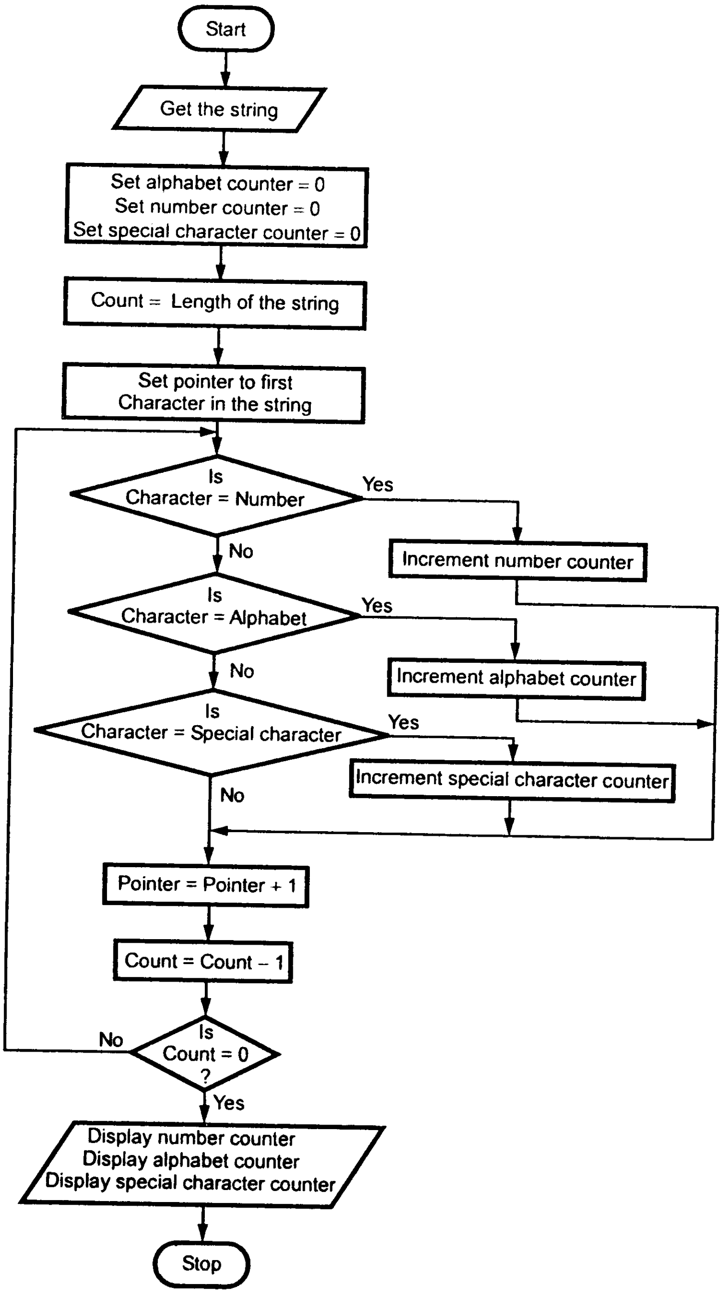
Flowchart :



Program 13 : Search numbers, alphabets, special characters

(Softcopy of this program, P20.asm is available at www.vtubooks.com)

Flowchart :



```
.MODEL SMALL
.STACK 100
TITLE    TOTAL
; (THIS PROGRAM GIVES THE TOTAL NUMBERS, ALPHABETS, SPECIAL
; CHARACTERS IN THE GIVEN STRING)
.DATA
        BUF      DB 80                    ;(MAX LENGTH OF ARRAY)
                 DB 00                    ;(ACTUAL LENGTH OF ARRAY)
                 DB 80 DUP (0)            ;(STARTING OF ARRAY)
    STR1         DB 10,13,'ENTER THE STRING:$'
    STR2         DB 10,13,'TOTAL NO:$'
    STR3         DB 10,13,'TOTAL ALPHABETS:$'
    STR4         DB 10,13,'TOTAL SPECIAL CHAR:$'
        NUM      DB 0
        SPC      DB 0
        ALPHA    DB 0

.CODE
START:  MOV      AX,@data             ; [ Initialise
        MOV      DS,AX                ;   data segment ]
        MOV      AH,09H
        MOV      DX,OFFSET STR1       ; Address of STR1
        INT      21H                  ; Display message STR1
        MOV      AH,0AH
        MOV      DX,OFFSET BUF        ; Get address of the buffer
        INT      21H                  ; Input the string
        MOV      BX,OFFSET BUF        ; Get address of the buffer
        INC      BX                   ; Increment address of buffer
        MOV      DL,[BX]              ; Get the length of string
        INC      BX                   ; Get the starting of array
NEXT:   MOV      AL,[BX]              ; Read the character
        CMP      AL,30H               ; Check for special character
        JB       INCSPC               ; If yes goto INCSPC
        CMP      AL,3AH               ; Check for number
        JB       INCNUM               ; If number goto INCNUM
        CMP      AL,41H               ; Check for special character
        JB       INCSPC               ; If yes goto INCSPC
        CMP      AL,5BH               ; Check for alphabet
        JB       INALP                ; If yes goto INALP
        CMP      AL,61H               ; Check for special character
        JB       INCSPC               ; If yes goto INCSPC
        CMP      AL,7BH               ; Check for alphabet
        JB       INALP                ; If yes goto INALP
INCSPC: MOV      AL,SPC
        ADD      AL,01H               ; [ INCR special character
                                      ;   counter and
        DAA                           ;     adjust it to decimal ]
        MOV      SPC,AL
        INC      BX                   ; Increment pointer to point
                                      ; the next character
```

```
         DEC     DL               ; Decrement counter
         JNZ     NEXT
         JMP     DISPLY           ; Otherwise goto DISPLY
INCNUM:  MOV     AL,NUM
         ADD     AL,01H           ; [ Increment number counter
         DAA                      ;  and adjust it to decimal ]
         MOV     NUM,AL
         INC     BX               ; Increment pointer to point
                                  ; the next character
         DEC     DL               ; Decrement counter
         JNZ     NEXT             ; If count not = 0, repeat
         JMP     DISPLY           ; Otherwise goto DISPLY
INALP:   MOV     AL,ALPHA
         ADD     AL,01H           ; [ Increment alphabet counter
         DAA                      ;  and adjust it to decimal ]
         MOV     ALPHA,AL
         INC     BX               ; Increment pointer to point
                                  ; the next character
         DEC     DL               ; Decrement counter
         JNZ     NEXT             ; If count not = 0, repeat
         JMP     DISPLY           ; Otherwise goto DISPLY
DISPLY:  MOV     DX,OFFSET STR2   ; Get the address of STR2
         MOV     AH,09H
         INT     21H              ; Display message STR2
         MOV     AL,NUM           ; Read the number count
         AND     AL,0F0H          ; Get MS digit in AL rotate AL
         MOV     CL,04H           ; Four times
         ROR     AL,CL
         ADD     Al,30H           ; Convert to ASCII
         MOV     DL,AL
         MOV     AH,02H           ; Display the MS digit
         INT     21H
         MOV     AL,NUM           ; Read the number count
         AND     AL,0FH           ; Get LS digit in AL
         ADD     AL,30H           ; Convert to ASCII
         MOV     DL,AL
         INT     21H              ; Display the LS digit
         MOV     DX,OFFSET STR3   ; Get address of STR3
         MOV     AH,09H
         INT     21H              ; Display message STR3
         MOV     AL,ALPHA         ; Read the alphabet count
         AND     AL,0F0H          ; Get MS digit in AL rotate AL
         MOV     CL,04H           ; Four times
         ROR     AL,CL
         ADD     AL,30H           ; Convert to ASCII
         MOV     DL,AL
         MOV     AH,02H
         INT     21H              ; Display the MS digit
         MOV     AL,ALPHA         ; Read the alphabet count
         AND     AL,0FH           ; Get LS digit in AL
```

```
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the LS digit
        MOV     DX,OFFSET STR4  ; Get the address of STR4
        MOV     AH,09H
        INT     21H             ; Display message STR4
        MOV     AL,SPC          ; Read the special character
                                ; count
        AND     AL,0F0H         ; Get MS digit in AL rotate AL
        MOV     CL,04           ; Four times
        ROR     AL,CL
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the MS digit
        MOV     AL,SPC          ; Read the special character count
        AND     AL,0FH          ; Get LS digit in AL
        ADD     AL,30H          ; Convert to ASCII
        MOV     DL,AL
        MOV     AH,02H
        INT     21H             ; Display the LS digit
        MOV     AH,4CH          ; [ Terminate and
        INT     21H             ;    Exit to DOS ]
        END     START
```

Program 14 : Program to find whether string is palindrome or not

(Softcopy of this program, P21.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.DATA
        M1      DB 10, 13, 'Enter the string : $'
        M2      DB 10, 13, 'String is palindrome $'
        M3      DB 10, 13, 'String is not palindrome $'
        BUFF    DB 80
                DB 0
                DB 80 DUP (0)
.CODE
START:  MOV AX,@data        ; [ Initialise
        MOV DS,AX           ;   data segment ]
        MOV AH,09H
        MOV DX,OFFSET M1
        INT 21H             ; Display message M1
        MOV AH,0AH          ; Input the string
        LEA DX,BUFF
        INT 21H
        LEA BX,BUFF+2       ; Get starting address of string
        MOV CH,00H
        MOV CL,BUFF+1
        MOV DI,CX
```

```
        DEC  DI
        SAR  CL,1
        MOV  SI,00H
BACK:   MOV  AL,[BX + DI]     ; Get the right most character
        MOV  AH,[BX + SI]     ; Get the left most character
        CMP  AL,AH            ; Check for palindrome
        JNZ  LAST             ; If not exit
        DEC  DI               ; Decrement end pointer
        INC  SI               ; Increment starting pointer
        DEC  CL               ; Decrement counter
        JNZ  BACK             ; If count not = 0, repeat
        MOV  AH,09H           ; Display message 2
        MOV  DX,OFFSET M2
        INT  21H
        JMP  TER
LAST:   MOV  AH,09H
        MOV  DX,OFFSET M3     ; Display message 3
        INT  21H
TER:    MOV  AH,4CH           ; [ Terminate and
        INT  21H              ;   Exit to DOS ]
        END  START
```

Program 15 : Program to display string in lowercase

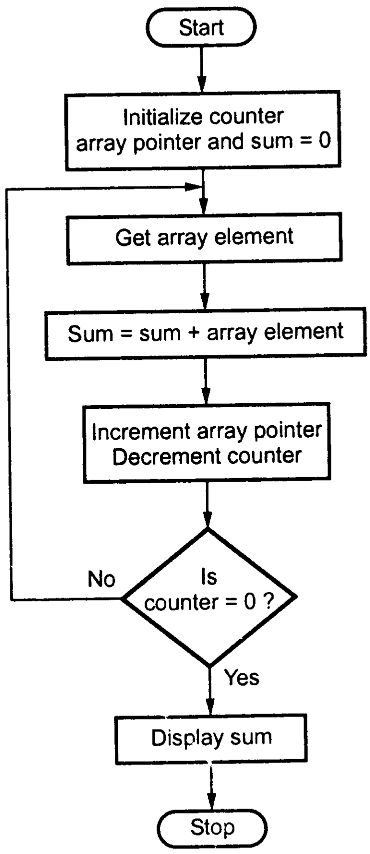
(Softcopy of this program, P22.asm is available at www.vtubooks.com)

```
.MODEL SMALL
.DATA
        M1      DB  10, 13, 'ENTER THE STRING : $'
        M2      DB  10, 13, 'THE LOWERCASE STRING : $'
        BUFF    DB  80
                DB  0
                DB  80 DUP (0)
.CODE
START : MOV  AX,@data         ; [ Initialise
        MOV  DS,AX            ;   data segment ]
        MOV  AH,09H           ; Display message1
        MOV  DX,OFFSET M1
        INT  21H
        MOV  AH,09H
        MOV  DX,OFFSET M2     ; Display message M2
        INT  21H
        MOV  AH,0AH           ; Input the string
        LEA  DX,BUFF
        INT  21H
        MOV  CH,00H
```

```
        MOV CL,BUFF+1          ; Take character count in CX
        LEA BX,BUFF+2
        MOV DI,00H
BACK :  MOV DL,[BX+DI]         ; point to the first character
        ADD DL,20H             ; convert to lowercase
        MOV AH,02H
        INT 21H                ; Display the character
        INC DI
        DEC CX                 ; Decrement character counter
        JNZ BACK               ; If not = 0, repeat
        MOV AH,4CH             ; [ Terminate and
        INT 21H                ;   Exit to DOS ]
        END START
```

Program 16 : Write an 8086 assembly language program (ALP) to add array of N number stored in the memory.

Flowchart :



Algorithm :

1. Initialize counter = N
2. Initialize array pointer.
3. Sum = 0
4. Get the array element pointed by array pointer.
5. Add array eiement in the sum.
6. Increment array pointer decrement counter.
7. Repeat steps 4, 5 and 6 until counter equal to zero.
8. Display sum.
9. Stop.

Sum of array having HEX numbers

```
        PAGE      52,80
        TITLE     8086 ALP to find sum of numbers in the array.
.MODEL  SMALL
.DATA
    ARRAY     DB 10H,20H,30H,40H,50H,60H,70H,80H,90H,00H
    SUM       DW 0
    MES       DB 10,13, 'Sum of array elements is : $'
.CODE
START:  MOV AX,@data        ; [ Initialise
        MOV DS,AX           ;   data segment ]
        MOV CL,10           ; Initialise counter
        XOR DI,DI           ; Initialise pointer
        LEA BX,ARRAY        ; Initialise array base pointer
BAC:    MOV AL,[BX+DI]      ; Get the number
        MOV AH,00H          ; Make higher byte 00h
        ADD SUM,AX          ; SUM = SUM + number
        INC DI              ; Increment pointer
        DEC CL              ; Decrement counter
        JNZ BAC             ; if not 0 go to back
        MOV AX,SUM          ; Get sum in AX
        CALL D_HEX          ; Display sum of array
        MOV AH, 4CH
        INT 21H
```

```
D_HEX PROC NEAR

       PUSH DX              ; Save registers
       PUSH CX
       PUSH AX

         MOV CL, 04H           ; Load rotate count
         MOV CH, 04H           ; Load digit count
BACK:    ROL AX, CL            ; rotate digits
         PUSH AX               ; save contents of AX
         AND AL, 0FH           ; [Convert
         CMP AL,9              ;  number
         JBE ADD30             ; to
         ADD AL, 37H           ; its
         JMP DISP              ; ASCII
ADD30:   ADD AL,30H            ; equivalent]
DISP:    MOV AH,02H
         MOV DL,AL             ; [Display the
         INT 21H               ;  number]
         POP AX                ; restore contents of AX
         DEC CH                ; decrement digit count
         JNZ BACK              ; if not zero repeat

         POP AX                ; Restore registers
         POP CX
         POP DX

         RET
         ENDP
         END
```

Sum of array having decimal numbers

```
    PAGE      52,80
    TITLE     8086 ALP to find sum of numbers in the array.
.MODEL SMALL
.DATA
```

```
    ARRAY     DB 12,24,26,63,25,86,20,33,10,35
    SUM       DW 0
    MES       DB 10,13, 'Sum of array elements is : $'
.CODE
START:  MOV AX,@data       ; [ Initialise
        MOV DS,AX          ;    data segment ]
        MOV CL,10          ; Initialise counter
        XOR DI,DI          ; Initialise pointer
        LEA BX,ARRAY       ; Initialise array base pointer
BAC:    MOV AL,[BX+DI]     ; Get the number
        MOV AH,00H         ; Make higher byte 00h
        ADD SUM,AX         ; SUM = SUM + number
        INC DI             ; Increment pointer
        DEC CL             ; Decrement counter
        JNZ BAC            ; if not 0 go to back
        MOV AX, SUM        ; Get the result
        CALL ATB4D         ; Display sum of array
        MOV AH, 4CH
        INT 21H

ATB4D PROC NEAR

        PUSH    DX         ; Save registers
        PUSH    CX
        PUSH    BX
        PUSH    AX

        MOV     CX, 0      ; Clear digit counter
        MOV     BX, 10     ; Load 10 decimal in BX
BACK :  MOV     DX, 0      ; Clear DX
        DIV     BX         ; divide DX : AX by 10
        PUSH    DX         ; Save remainder
        INC     CX         ; Counter remainder
        OR      AX, AX     ; test if quotient equal to zero
        JNZ BACK           ; if not zero divide again
        MOV AH, 02H        ; load function number
```

```
DISP:   POP DX          ; get remainder
        ADD DL, 30H     ; Convert to ASCII
        INT 21H         ; display digit
        LOOP DISP

        POP AX          ; Restore registers
        POP BX
        POP CX
        POP DX

        RET
        ENDP
        END
```

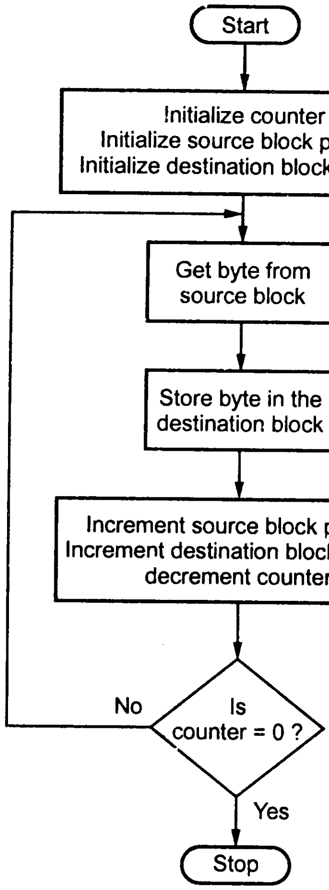
Program 17 : Write 8086 ALP to perform non-overlapped and overlapped block transfer.

In non-overlapped block transfer, source block and destination blocks are different. Here, we can transfer byte-by-byte or word-by-word data from one block to another block.

Algorithm :

1. Initialize counter.
2. Initialize source block pointer.
3. Initialize destination block pointer.
4. Get the byte from source block.
5. Store the byte in the destination block.
6. Increment source, destination pointers and decrement counter.
7. Repeat steps 4, 5 and 6 unit counter equal to zero.
8. Stop.

Flowchart :



Non-overlapped block transfer

```
          PAGE      52,80
          TITLE     Non overlapped block transfer.
.MODEL    SMALL
.STACK    100
.DATA
     ARRAY              DB 12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H
     NEW_ARR            DB 10 DUP (?)
.CODE
START:    MOV AX,@data       ; [ Initialise
          MOV DS,AX          ; data segment and
          MOV ES,AX          ; extra segment ]
          MOV CX,10          ; Initialise counter
          LEA SI,ARRAY       ; Initialise source_pointer
```

```
          LEA     DI,NEW_ARR    ; Initialise destination_pointer
          CLD                   ; Clear direction flag to
                                ; autoincrement SI and DI
          MOV     AL,[SI]       ; [Get the number
          MOV     [DI],AL       ; and save number in new array ]
          REP     MOVSB         ; Decrement CX and MOVSB until CX
                                  will be 0

          LEA     DI,NEW_ARR    ; Initialise destination_pointer
          MOV     CX,10         ; Initialize counter
BACK1:    MOV     AH,[DI]       ; Get number
          CALL    D_HEX2        ; Display number
          CALL    SPACE         ; Display space
          INC     DI            ; Increment destination_pointer
          LOOP    BACK1         ; if counter not zero, repeat
          MOV     AH,4CH        ; Return to DOS
          INT     21H

D_HEX2  PROC NEAR
          PUSH    CX
          MOV     CL, 04H       ; Load rotate count
          MOV     CH, 02H       ; Load digit count
BAC:      ROL     AX, CL        ; rotate digits
          PUSH    AX            ; save contents of AX
          AND     AL, 0FH       ; [Convert
          CMP     AL,9          ;  number
          JBE     Add30         ; to
          ADD     AL, 37H       ; its
          JMP     DISP          ; ASCII
Add30:    ADD     AL,30H        ; equivalent]
DISP:     MOV     AH,02H
          MOV     DL,AL         ; [Display the
          INT     21H           ;  number]
          POP     AX            ; restore contents of AX
          DEC     CH            ; decrement digit count
          JNZ     BAC           ; if not zero repeat
          POP     CX
          RET
          ENDP
```

```
SPACE PROC NEAR
      PUSH    AX           ; Save registers
      PUSH    DX
      MOV     AH,02        ; Display space
      MOV     DL,' '
      INT     21H
      POP     DX           ; restore registers
      POP     AX
      RET                  ; return to main program
      ENDP
      END
```

Overlapped block transfer

We call two blocks are overlapped when some portion of source and destination blocks are common. As shown in the Fig. 5.1, source and destination blocks can be overlapped in two ways. In first case Fig. 5.1 (a) we can begin transfer from starting location of source block to the starting location of destination block, i.e. [20000H] ← [20005H]

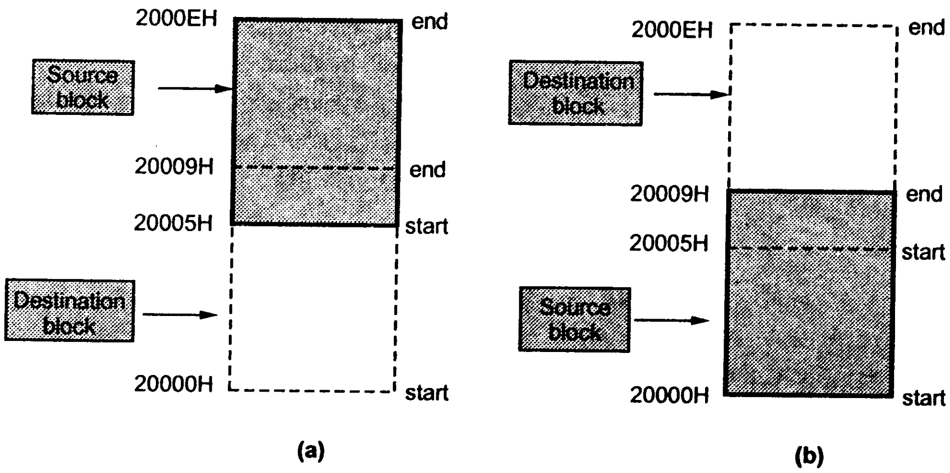


Fig. 5.1

We can then increment source and destination block pointers and carry on byte transfer until the pointers reach the end of two blocks, i.e. upto [20009H] ← [2000EH].

In second case Fig. 5.1 (b) we cannot use the same block transfer procedure, because there will be over writing of data within the source block, i.e. at first byte transfer contents of 20000H will be over written in the location 20005H and data at 20005H in the source block get lost. To avoid over writing in such cases we have to transfer data from source block to destination block from the end of the block, i.e. we have to begin with the transfer [2000EH] ← [20009H], decrement the source and destination pointers and carry on the byte transfer until the pointer reach the start of the blocks, i.e. upto [20005H] ← [20000H]

```
        PAGE     52,80
        TITLE    Overlapped block transfer.
    .MODEL SMALL
    .STACK 100
    .DATA
        ARRAY   DB,12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H,?,?,?,?,?
    .CODE
    START:     MOV  AX,@data        ; [ Initialise
               MOV  DS,AX           ; data segment and
               MOV  ES,AX           ; extra segment ]
               MOV  CX,10           ; Initialise counter
               LEA  SI,ARRAY+9      ; Initialise source_pointer
               LEA  DI,ARRAY+14     ; Initialise destination_pointer
               STD                  ; SET direction flag to
                                      autodecrement SI and DI
               MOV  AL,[SI]         ; Get the number
               MOV  [DI],AL         ; and save number in new array ]
               REP  MOVSB           ; Decrement CX and MOVSB until
                                    ; CX will be 0

               LEA  DI,ARRAY+5      ; Initialise destination_pointer
               MOV  CX,10           ; Initialize counter
    BACK1:     MOV  AH,[DI]         ; Get number
               CALL D_HEX2          ; Display number
               CALL SPACE           ; Display space
               INC  DI              ; Increment destination_pointer
               LOOP BACK1           ; If counter not zero repeat
               MOV  AH,4CH          ; Return to DOS
               INT  21H
```

```
D_HEX2 PROC NEAR
        PUSH    CX
        MOV     CL, 04H         ; Load rotate count
        MOV     CH, 02H         ; Load digit count
BAC:    ROL     AX, CL          ; rotate digits
        PUSH    AX              ; save contents of AX
        AND     AL, 0FH         ; [Convert
        CMP     AL,9            ;  number
        JBE     Add30           ; to
        ADD     AL, 37H         ; its
        JMP     DISP            ; ASCII
Add30:  ADD     AL,30H          ; equivalent]
DISP:   MOV     AH,02H
        MOV     DL,AL           ; [Display the
        INT     21H             ;  number]
        POP     AX              ; restore contents of AX
        DEC     CH              ; decrement digit count
        JNZ     BAC             ; if not zero repeat
        POP     CX
        RET
        ENDP
SPACE PROC NEAR
        PUSH    AX              ; save registers
        PUSH    DX
        MOV     AH,02           ; display space
        MOV     DL,' '
        INT     21H
        POP     DX              ; restore registers
        POP     AX
        RET                     ; return to main program
        ENDP
        END
```

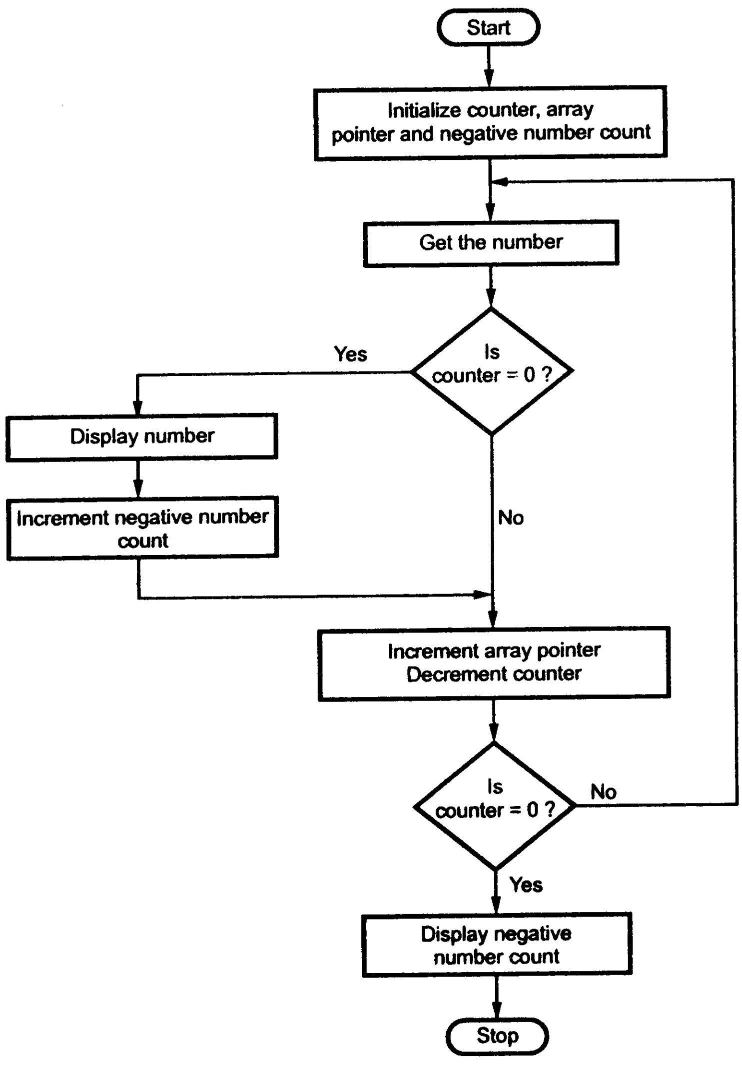
Program 18 : Write 8086 ALP to find and count negative numbers from the array of signed numbers stored in memory.

In sign number representation, number is called negative when its most significant bit (MSB) is 1. This bit can be checked by masking all other bits with the help of logical AND instruction.

Algorithm :

1. Initialize counter.
2. Initialize array pointer.
3. Initialize negative number count.
4. Get the number.
5. Check sign of number by checking its MSB. If negative increment negative number count and display the number.
6. Decrement counter and increment array pointer.
7. Repeat steps 4, 5 and 6 until counter equal to zero.
8. Display negative number count.
9. Stop.

Flowchart :



```
        PAGE      52,80
        TITLE     Find and count the negative numbers in the array.
        .MODEL SMALL
        .STACK 100
        .DATA
             ARRAY     DB 92H,23H,96H,0A3H,25H,86H,2FH,33H,10H,35H
             MES       DB 10,13, 'Negative numbers are : $'
             MES1      DB 10,13, 'Total Negative number count is : $'
        .CODE
START:       MOV       AX,@data       ; [ Initialise
             MOV       DS,AX          ; data segment ]
             MOV       CX,10          ; Initialise counter
             MOV       BH,0           ; Initialise negative number count
                                        equal to 0
             LEA       BP,ARRAY       ; Initialise array base_pointer
             LEA       DX, MES
             MOV       AH, 09H
             INT       21H
BACK:        MOV       AL,DS:[BP]     ; Get the number
             MOV       AH,AL          ; Save number in AH
             AND       AL,80H         ; Mask all bits except MSB
             JZ        NEXT           ; If MSB = 0 go to next
             CALL D_HEX2              ; Otherwise display number
             CALL SPACE
             INC       BH             ; Increment negative number count
NEXT :       INC       BP             ; Increment array base_pointer
             LOOP BACK                ; Decrement counter
                                      ; if not 0 go to back
             LEA       DX, MES1
             MOV       AH, 09H
             INT       21H
             MOV       AH,02H
             ADD       BH,30H
             MOV       DL,BH
             INT       21H
             MOV       AH,4CH         ; [ Exit
             INT       21H            ;   to DOS ]
```

```
D_HEX2 PROC NEAR
        MOV     CL, 04H         ; Load rotate count
        MOV     CH, 02H         ; Load digit count
BAC:    ROL     AX, CL          ; rotate digits
        PUSH    AX              ; save contents of AX
        AND     AL, 0FH         ; [Convert
        CMP     AL,9            ;  number
        JBE     Add30           ; to
        ADD     AL, 37H         ; its
        JMP     DISP            ; ASCII
Add30:
        ADD     AL, 30H         ; equivalent]
DISP:   MOV     AH, 02H
        MOV     DL, AL          ; [Display the
        INT     21H             ;  number]
        POP     AX              ; restore contents of AX
        DEC     CH              ; decrement digit count
        JNZ     BAC             ; if not zero repeat
        ENDP
SPACE PROC NEAR
    PUSH    AX                  ; save AX
    MOV  AH,02H                 ; [ Call DOS routine
    MOV  DL, ' '                ;   to leave space ]
    INT  21H                    ; restore AX
    POP  AX                     ; return to main program
    RET
    ENDP

    END
```

Program 19 : Convert BCD to HEX and HEX to BCD

Write 8086 ALP to convert 4-digit HEX number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choices from user for :

a. HEX to BCD

b. BCD to HEX

c. EXIT

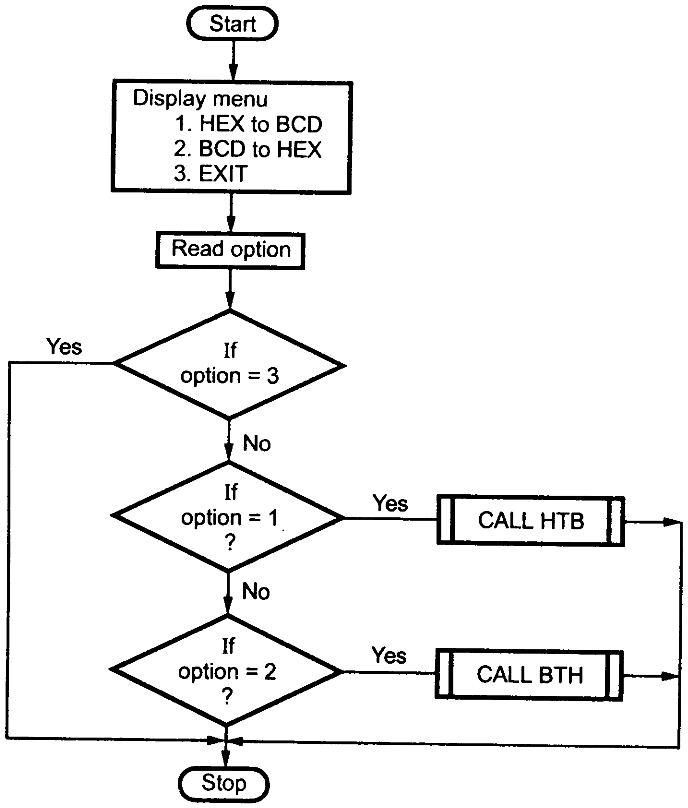
Display proper strings to prompt the user while accepting the input and displaying the result.

In this program we use the standard routines explained in the chapter 3 to convert data from one form to other. However, to select the conversion we display menu on the screen and display proper messages on the screen to guide user. Therefore, in this program separate macro named PROMPT is written for display the message. After accepting the option from the user, the option is checked and proper routine is called to perform desired operation.

Algorithm :

1. Display menu
   a. HEX To BCD
   b. BCD To HEX
   c. EXIT
   ENTER THE CHOICE :
2. Read the option
   It option is 3-exit
   1 - Do HEX to BCD conversion
   2 - Do BCD to HEX conversion
3. Stop

Flowchart :



```
PROMPT MACRO MESSAGE        ; Define macro with MESSAGE as a
                            ; parameter
        PUSH AX             ; Save AX register
        MOV  AH, 09H        ; display message
        LEA  DX, MESSAGE
        INT  21H
        POP  AX             ; restore register
        ENDM

.MODEL SMALL        ; select small model
.STACK 100

.DATA               ; start data segment
    MES1      DB 10, 13, '1. HEX TO BCD $'
    MES2      DB 10, 13, '2. BCD TO HEX $'
    MES3      DB 10, 13, '3. EXIT $'
    MES4      DB 10, 13, 'ENTER THE CHOICE : $'
    MES5      DB 10, 13, 'ENTER CORRECT CHOICE : $'
    MES6      DB 10, 13,'$'
    MES7      DB 10, 13, 'ENTER THE FOUR DIGIT HEX NUMBER : $'
    MES8      DB 10, 13, 'EQUIVALENT BCD NUMBER IS : $'
    MES9      DB 10, 13, 'ENTER THE BCD NUMBER : $'
    MES10     DB 10, 13, 'EQUIVALENT HEX NUMBER IS : $'

        NUMBER DW ?         ; define NUMBER
.CODE                       ; start code segment
START:  MOV AX, @DATA       ; [Initialize
        MOV DS, AX          ;  data segment]

        PROMPT MES1         ; Display MES1
        PROMPT MES2         ; Display MES2
        PROMPT MES3         ; Display MES3
        PROMPT MES4         ; Display MES4
```

```
AGAIN:   MOV      AH,01     ; [ READ
         INT      21H       ;   OPTION ]

         PROMPT   MES6      ; Display MES6

         CMP      AL,'3'    ; [ If choice is 3
         JZ       LAST      ;   exit ]

         CMP      AL,'1'    ; [ If choice is 1
         JNZ      NEXT1
         CALL     HTB       ;   Do HEX to BCD conversion
         JMP      LAST      ;   exit ]

NEXT1:   CMP      AL,'2'    ; [ If choice is 2
         JNZ      NEXT2
         CALL     BTH       ;   Do BCD to HEX conversion
         JMP      LAST      ;   exit ]

NEXT2:   PROMPT   MES5      ; Display MES5
         JMP      AGAIN

LAST:    MOV      AH,4CH    ; Return to DOS
         INT      21H

HTB PROC NEAR

         PROMPT MES7
         CALL R_HEX
         PROMPT MES8
         CALL D_BCD
         RET
         ENDP
```

```
BTH PROC NEAR
        PROMPT   MES9
        MOV      CX, 10        ; load 10 decimal in CX
        MOV      BX, 0         ; clear result
BACK2:  MOV      AH,01H        ;[Read key
        INT      21H           ; with echo]
        CMP      AL,'0'
        JB       SKIP          ; jump if below '0'
        CMP      AL,'9'
        JA       SKIP          ; jump if above '9'
        SUB      AL, 30H       ; convert to BCD
        PUSH     AX            ; save digit
        MOV      AX, BX        ; multiply previous result by 10
        MUL      CX
        MOV      BX, AX        ; get the result in BX
        POP      AX            ; retrieve digit
        MOV      AH, 00H
        ADD      BX, AX        ; Add digit value to result
        JMP      BACK2         ; Repeat
SKIP :  MOV      AX,BX         ; save the result in AX
        PROMPT   MES10
        CALL     D_HEX
        RET
        ENDP
```

```
R_HEX PROC NEAR
    MOV CL, 04          ;  load shift count
    MOV SI, 04          ;  load iteration count
    MOV BX, 0           ;  clear result
BAC: MOV AH, 01         ; [Read a key
    INT 21H             ;  with echo]

    CALL CONV           ;  convert to binary

    SHL BX, CL          ; [pack four
    ADD BL, AL          ;  binary digits
    DEC SI              ;  as 16-bit
    JNZ BAC             ;  number]
    MOV NUMBER, BX      ; save result at NUMBER
    ENDP
```

```
; The procedure to convert contents of AL into
; hexadecimal equivalent
    CONV PROC NEAR

         CMP  AL,'9'
         JBE  SUBTRA30        ; if number is between 0 through 9
         CMP  AL,'a'
         JB   SUBTRA37        ; if letter is uppercase
         SUB  AL, 57H         ; subtract 57H if letter is lowercase
                                JMP LAST1
SUBTRA30:SUB  AL, 30H         ; convert number
                                JMP LAST1
SUBTRA37:SUB  AL, 37H         ; convert uppercase letter
LAST1:   RET
CONV     ENDP
```

```
D_BCD PROC NEAR

    MOV AX,NUMBER
    MOV CX, 0           ; Clear digit counter
    MOV BX, 10          ; Load 10 decimal in BX
BACK:MOV DX, 0          ; Clear DX
    DIV BX              ; divide DX : AX by 10
    PUSH DX             ; Save remainder
    INC CX              ; Counter remainder
    OR  AX, AX          ; test if quotient equal to zero
    JNZ BACK            ; if not zero divide again
    MOV AH, 02H         ; load function number
DISP:POP DX             ; get remainder
    ADD DL, 30H         ; Convert to ASCII
    INT 21H             ; display digit
    LOOP DISP
    RET
    ENDP
```

```
D_HEX PROC NEAR
         MOV CL, 04H        ; Load rotate count
         MOV CH, 04H        ; Load digit count
BAC1:    ROL AX, CL         ; rotate digits
         PUSH AX            ; save contents of AX
```

```
          AND AL, 0FH          ; [Convert
          CMP AL,9             ;  number
          JBE Add30            ; to
          ADD AL, 37H          ; its
          JMP DISP1            ; ASCII
Add30:
          ADD AL,30H           ; equivalent]
DISP1:    MOV AH,02H
          MOV DL,AL            ; [Display the
          INT 21H              ;  number]
          POP AX               ; restore contents of AX
          DEC CH               ; decrement digit count
          JNZ BAC1             ; if not zero repeat
          RET
          ENDP

          END
```

Program 20 : Multiplication of two 8-bit numbers

Algorithm :

1. Read 2-digit hex number as a multiplicand.
2. Read 2-digit hex number as a multiplier.
3. Initialize iteration count = 8 since multiplier is 8-bit.
4. Make result = 0.
5. Shift result left by 1-bit.
6. Rotate multiplier 1-bit to check current MSB if bit is 1, Add multiplicand in the result.
7. Decrement iteration count and repeat steps 5 and 6 fill iteration count is zero.
8. Display result.
9. Stop.