MODULE – 2

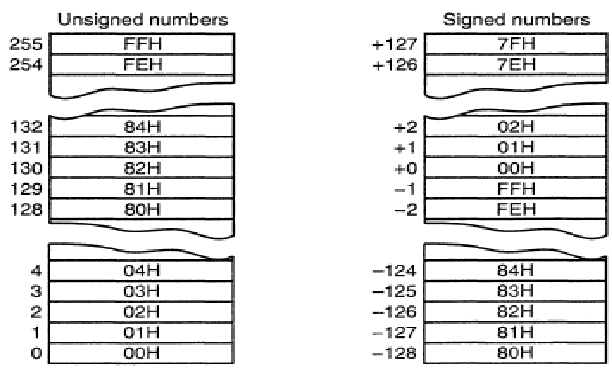
A AND L INSTRUCTIONS & INT 21H AND INT 10H PROGRAMMING

ARITHMETIC & LOGIC INSTRUCTIONS AND PROGRAMS

INTRUCTIONS SET DESCRIPTION:

UNSIGNED ADDITION AND SUBTRACTION:

Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign. This means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data, and between 0000 and FFFFH (0 to 65535 decimal) for 16-bit data.



Addition of Unsigned Numbers:

```
ADD destination,source ;destination = destination + source
```

- ✓ The instructions ADD and ADC are used to add two operands. The destination operand can be a register or in memory. The source operand can be a register, in memory, or immediate.
- ✓ Remember that memory-to-memory operations are never allowed in x86 Assembly language.
- ✓ The instruction could change any of the ZF, SF, AF, CF, or PF bits of the flag register, depending on the operands involved. The overflow flag is used only in signed number operations.

Show how the flag register is affected by

```
MOV  AL,0F5H
ADD  AL,0BH
```

Solution:

```
   F5H        1111 0101
+  0BH     +  0000 1011
  100H        0000 0000
```

After the addition, the AL register (destination) contains 00 and the flags are as follows:
CF = 1, since there is a carry out from D7
SF = 0, the status of D7 of the result
PF = 1, the number of 1s is zero (zero is an even number)
AF = 1, there is a carry from D3 to D4
ZF = 1, the result of the action is zero (for the 8 bits)

With addition, two cases will be discussed:

CASE1: Addition of Individual Byte and Word Data:

Write a program to calculate the total sum of 5 bytes of data. Each byte represents the daily wages of a worker. This person does not make more than $255 (FFH) a day. The decimal data is as follows: 125, 235, 197, 91, and 48.

```
TITLE       PROG3-1A  (EXE)  ADDING 5 BYTES
PAGE        60,132
.MODEL SMALL
.STACK 64
;--------------------------------
            .DATA
COUNT       EQU    05
DATA        DB           125,235,197,91,48
            ORG    0008H
SUM         DW     ?
;-------------------------------
      .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    CX,COUNT           ;CX is the loop counter
     MOV    SI,OFFSET DATA     ;SI is the data pointer
     MOV    AX,00              ;AX will hold the sum
BACK:ADD    AL,[SI]            ;add the next byte to AL
     JNC    OVER         ;if no carry, continue
     INC    AH           ;else accumulate carry in AH
OVER:INC    SI           ;increment data pointer
     DEC    CX           ;decrement loop counter
     JNZ    BACK         ;if not finished, go add next byte
     MOV    SUM,AX       ;store sum
     MOV    AH,4CH
     INT    21H          ;go back to OS
MAIN ENDP
     END    MAIN
```

Program 3-1a

These numbers are converted to hex by the assembler as follows: 125 = 7DH, 235 = 0EBH, 197 = 0C5H, 91 = 5BH, 48 = 30H. This program uses AH to accumulate carries as the operands are added to AL register. Three iterations of the loop are shown below:

1. In the first iteration of the loop, 7DH is added to AL with CF = 0 and AH = 00. CX = 04 and ZF = 0.
2. In the second iteration of the loop, EBH is added to AL, which results in AL = 68H and CF = 1. Since a carry occurred, AH is incremented. CX = 03 and ZF = 0.
3. In the third iteration, C5H is added to AL, which makes AL = 2DH. Again a carry occurred, so AH is incremented again. CX = 02 and ZF = 0.

This process continues until CX = 00 and the zero flag becomes 1, which will cause JNZ to fall through. Then the result will be saved in the word-sized memory set aside in the data segment.

Although this program works correctly, due to pipelining it is strongly recommended that the following lines of the program be replaced:

Replace these lines	With these lines
`BACK: ADD AL,[ SI]`	`BACK: ADD AL,[ SI]`
`JNC OVER`	`ADC AH,00 ;add 1 to AH if CF=1`
`INC AH`	`INC SI`
`OVER: INC SI`	

The instruction "*JNC OVER*" has to empty the queue of pipelined instructions and fetch the instructions from the OVER target every time the carry is zero (CF = 0). Hence, the "*ADC AH, 00*" instruction is much more efficient.

The addition of many word operands works the same way. Register AX (or CX, DX, or BX) could be used as the accumulator and BX (or any general-purpose 16-bit register) for keeping the carries. Program 3-1b is the same as Program 3-1a, rewritten for word addition.

Write a program to calculate the total sum of five words of data. Each data value represents the yearly wages of a worker. This person does not make more than $65,555 (FFFFH) a year. The decimal data is as follows: 27345, 28521, 29533, 30105, and 32375.

```
TITLE      PROG3-1B (EXE) ADDING 5 WORDS
PAGE       60,132
.MODEL SMALL
.STACK 64
;-----------------
       .DATA
COUNT  EQU        05
DATA   DW         27345,28521,29533,30105,32375
       ORG        0010H
SUM    DW         2 DUP(?)
;-----------------
     .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    CX,COUNT          ;CX is the loop counter
     MOV    SI,OFFSET DATA    ;SI is the data pointer
     MOV    AX,00             ;AX will hold the sum
     MOV    BX,AX             ;BX will hold the carries
BACK:ADD    AX,[ SI]          ;add the next word to AX
     ADC    BX,0        ;add carry to BX
     INC    SI          ;increment data pointer twice
     INC    SI          ;to point to next word
     DEC    CX          ;decrement loop counter
     JNZ    BACK        ;if not finished, continue adding
     MOV    SUM,AX            ;store the sum
     MOV    SUM+2,BX    ;store the carries
     MOV    AH,4CH
     INT    21H         ;go back to OS
MAIN ENDP
     END    MAIN
```

Program 3-1b

CASE2: Addition of Multiword Numbers:

```
TITLE      PROG3-2  (EXE)  MULTIWORD ADDITION
PAGE       60,132
.MODEL SMALL
.STACK 64
;-----------------------------------
      .DATA
DATA1 DQ    548FB9963CE7H
      ORG   0010H
DATA2 DQ    3FCD4FA23B8DH
      ORG   0020H
DATA3 DQ    ?
;-----------------------------------
     .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     CLC                         ;clear carry before first addition
     MOV    SI,OFFSET DATA1      ;SI is pointer for operand1
     MOV    DI,OFFSET DATA2      ;DI is pointer for operand2
     MOV    BX,OFFSET DATA3      ;BX is pointer for the sum
     MOV    CX,04                ;CX is the loop counter
BACK:MOV    AX,[SI]                    ;move the first operand to AX
     ADC    AX,[DI]                    ;add the second operand to AX
     MOV    [BX],AX                    ;store the sum
     INC    SI                         ;point to next word of operand1
     INC    SI
     INC    DI                   ;point to next word of operand2
     INC    DI
     INC    BX                   ;point to next word of sum
     INC    BX
     LOOP   BACK                 ;if not finished, continue adding
     MOV    AH,4CH
     INT    21H                  ;go back to OS
MAIN ENDP
     END    MAIN
```

Program 3-2

- Assume, a program is needed that will add the total Indian budget for the last 100 years or the mass of all the planets in the solar system.
- In cases like this, the numbers being added could be up to 8 bytes wide or even more. Since registers are only 16 bits wide (2 bytes), it is the job of the programmer to write the code to break down these large numbers into smaller chunks to be processed by the CPU.
- If a 16-bit register is used and the operand is 8 bytes wide, that would take a total of four iterations. However, if an 8-bit register is used, the same operands would require eight iterations.

✓ In writing this program, the first thing to be decided was the directive used for coding the data in the data segment. DQ was chosen since it can represent data as large as 8 bytes wide.

✓ In the addition of multibyte (or multiword) numbers, the ADC instruction is always used since the carry must be added to the next-higher byte (or word) in the next iteration. Before executing

ADC, the carry flag must be cleared (CF = 0) so that in the first iteration, the carry would not be added. Clearing the carry flag is achieved by the CLC (clear carry) instruction.

- ✓ Three pointers have been used: SI for DATA1, DI for DATA2, and BX for DATA3 where the result is saved.
- ✓ There is a new instruction in that program, "*LOOP xxxx*", which replaces the often used "*DEC CX*" and "*JNZ xxxx*".

```
LOOP   xxxx   ;is equivalent to  |  DEC   CX
                                 |  JNZ   xxxx
```

When "*LOOP xxxx*" is executed, CX is decremented automatically, and if CX is not 0, the microprocessor will jump to target address xxxx. If CX is 0, the next instruction (the one below "*LOOP xxxx*") is executed.

Subtraction of Unsigned Numbers:

```
SUB   dest,source;dest = dest - source
```

The x86 uses internal adder circuitry to perform the subtraction command. Hence, the 2's complement method is used by the microprocessor to perform the subtraction. The steps involved is –

1. Take the 2's complement of the subtrahend (source operand)
2. Add it to the minuend (destination operand)
3. Invert the carry.

These three steps are performed for every SUB instruction by the internal hardware of the x86 CPU. It is after these three steps that the result is obtained and the flags are set. The following example illustrates the three steps:

Show the steps involved in the following:

```
        MOV   AL,3FH        ;load AL=3FH
        MOV   BH,23H        ;load BH=23H
        SUB   AL,BH         ;subtract BH from AL. Place result in AL.
```

Solution:

```
 AL     3F            0011 1111          0011 1111
-BH    -23          - 0010 0011        + 1101 1101 (2's complement)
        1C                             1 0001 1100 CF=0 (step 3)
```

The flags would be set as follows: CF = 0, ZF = 0, AF = 0, PF = 0, and SF = 0.
The programmer must look at the carry flag (not the sign flag) to determine if the result is positive or negative.

- ✓ After the execution of SUB, if CF = 0, the result is positive; if CF = 1, the result is negative and the destination has the 2's complement of the result.

- Normally, the result is left in 2's complement, but the NOT and INC instructions can be used to change it. The NOT instruction performs the 1's complement of the operand; then the operand is incremented to get the 2's complement; as shown in the following example:

Analyze the following program:

```
;from the data segment:
DATA1       DB    4CH
DATA2       DB    6EH
DATA3       DB    ?
;from the code segment:
            MOV   DH,DATA1    ;load DH with DATA1 value (4CH)
            SUB   DH,DATA2    ;subtract DATA2 (6E) from DH (4CH)
            JNC   NEXT        ;if CF=0 jump to NEXT target
            NOT   DH          ;if CF=1 then take 1's complement
            INC   DH          ;and increment to get 2's complement
NEXT:       MOV   DATA3,DH    ;save DH in DATA3
```

Solution:

Following the three steps for "SUB DH,DATA2":

```
     4C     0100 1100         0100 1100
    -6E     0110 1110       + 1001 0010   (2's complement)
    -22                      01101 1110 CF=1 (step 3)result is negative
```

SBB (Subtract with Borrow):

This instruction is used for multibyte (multiword) numbers and will take care of the borrow of the lower operand. If the carry flag is 0, SBB works like SUB. If the carry flag is 1, SBB subtracts 1 from the result. Notice the "*PTR*" operand in the following Example.

Analyze the following program:

```
DATA_A      DD    62562FAH
DATA_B      DD    412963BH
RESULT      DD    ?
...         ...   ...
            MOV   AX,WORD PTR DATA_A       ;AX=62FA
            SUB   AX,WORD PTR DATA_B       ;SUB 963B from AX
            MOV   WORD PTR RESULT,AX       ;save the result
            MOV   AX,WORD PTR DATA_A +2    ;AX=0625
            SBB   AX,WORD PTR DATA_B +2    ;SUB 0412 with borrow
            MOV   WORD PTR RESULT+2,AX     ;save the result
```

Solution:

After the SUB, AX = 62FA – 963B = CCBF and the carry flag is set. Since CF = 1, when SBB is executed, AX = 625 – 412 – 1 = 212. Therefore, the value stored in RESULT is 0212CCBF.

The PTR (pointer) data directive is used to specify the size of the operand when it differs from the defined size. In above Example; "*WORD PTR*" tells the assembler to use a word operand, even though the data is defined as a double word.

UNSIGNED MULTIPLICATION AND DIVISION:

One of the major changes from the 8080/85 microprocessor to the 8086 was inclusion of instructions for multiplication and division. The use of registers AX, AL, AH, and DX is necessary.

Multiplication of Unsigned Numbers:

In discussing multiplication, the following cases will be examined: (1) byte times byte, (2) word times word, and (3) byte times word.

8-bit * 8-bit	AL * BL	16-bit * 16-bit	AX * BX
16-bit	AX	32-bit	DX AX

byte x byte: In byte-by-byte multiplication, one of the operands must be in the AL register and the second operand can be either in a register or in memory. After the multiplication, the result is in AX.

```
RESULT  DW   ?          ;result is defined in the data segment
        ...
        MOV  AL,25H     ;a byte is moved to AL
        MOV  BL,65H     ;immediate data must be in a register
        MUL  BL         ;AL = 25 x 65H
        MOV  RESULT,AX  ;the result is saved
```

In the program above, 25H is multiplied by 65H and the result is saved in word-sized memory named RESULT. Here, the register addressing mode is used.

The next three examples show the register, direct, and register indirect addressing modes.

```
;from the data segment:
DATA1       DB      25H
DATA2       DB      65H
RESULT      DW      ?
;from the code segment:
      MOV   AL,DATA1
      MOV   BL,DATA2
      MUL   BL                  ;register addressing mode
      MOV   RESULT,AX
or
      MOV   AL,DATA1
      MUL   DATA2               ;direct addressing mode
      MOV   RESULT,AX
or
      MOV   AL,DATA1
      MOV   SI,OFFSET DATA2
      MUL   BYTE PTR [SI]       ;register indirect addressing mode
      MOV   RESULT,AX
```

- ✓ In the register addressing mode example, any 8-bit register could have been used in place BL.
- ✓ Similarly, in the register indirect example, BX or DI could have been used as pointers.
- ✓ If the register indirect addressing mode is used, the operand size must be specified with the help of the PTR pseudo-instruction. In the absence of the "*BYTE PTR*" directive in the example above,

the assembler could not figure out if it should use a byte or word operand pointed at by SI. This confusion may cause an error.

word x word: In word-by-word multiplication, one operand must be in AX and the second operand can be in a register or memory. After the multiplication, registers DX and AX will contain the result. Since word-by-word multiplication can produce a 32-bit result, DX will hold the higher word and AX the lower word.

```
DATA3       DW    2378H
DATA4       DW    2F79H
RESULT1     DW    2 DUP(?)
...         ....
            MOV   AX,DATA3      ;load first operand into AX
            MUL   DATA4         ;multiply it by the second operand
            MOV   RESULT1,AX    ;store the lower word result
            MOV   RESULT1+2,DX  ;store the higher word result
```

word x byte: This is similar to word-by-word multiplication, except that AL-contains the byte operand and AH must be set to zero.

```
;from the data segment:
DATA5       DB    6BH
DATA6       DW    12C3H
RESULT3     DW    2 DUP(?)
;from the code segment:
            MOV   AL,DATA5      ;AL holds byte operand
            SUB   AH,AH         ;AH must be cleared
            MUL   DATA6         ;byte in AL mult. by word operand
            MOV   BX,OFFSET RESULT3 ;BX points to product
            MOV   [BX],AX           ;AX holds lower word
            MOV   [BX]+2,DX         ;DX holds higher word
```

Table: Unsigned Multiplication Summary

Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX
word × word	AX	register or memory	DX AX
word × byte	AL = byte, AH = 0	register or memory	DX AX

Division of Unsigned Numbers:

In the division of unsigned numbers, the following cases are discussed:

1. Byte over byte
2. Word over word
3. Word over byte
4. Double-word over word

<u>8-bit</u>	<u>AL</u>	Q: AL	<u>16-bit</u>	<u>AX</u>	Q: AX
8-bit	BL	R: AH	16-bit	BX	R: DX

16-bit	AX	Q: AL	32-bit	DA AX	Q: AX
8-bit	BL	R: AH	16-bit	BX	R: DX

In divide, there could be cases where the CPU cannot perform the division. In these cases an ***interrupt*** is activated. This is referred to as an *exception*. In following situations, the microprocessor cannot handle the division and must call an interrupt:

1. If the denominator is zero (dividing any number by 00)
2. If the quotient is too large for the assigned register.

In the IBM PC and compatibles, if either of these cases happens, the PC will display the "divide error" message.

byte/byte: In dividing a byte by a byte, the numerator must be in the AL register and AH must be set to zero. The denominator cannot be immediate but can be in a register or memory. After the DIV instruction is performed, the quotient is in AL and the remainder is in AH.

```
QOUT1       DB    ?
REMAIN1     DB    ?

;using immediate addressing mode will give an error
            MOV   AL,DATA7          ;move data into AL
            SUB   AH,AH             ;clear AH
            DIV   10                ;immed. mode not allowed!!
;allowable modes include:
;using direct mode
            MOV   AL,DATA7          ;AL holds numerator
            SUB   AH,AH             ;AH must be cleared
            DIV   DATA8             ;divide AX by DATA8
            MOV   QOUT1,AL          ;quotient = AL = 09
            MOV   REMAIN1,AH        ;remainder = AH = 05
;using register addressing mode
            MOV   AL,DATA7          ;AL holds numerator
            SUB   AH,AH             ;AH must be cleared
            MOV   BH,DATA8          ;move denom. to register
            DIV   BH                ;divide AX by BH
            MOV   QOUT1,AL          ;quotient = AL = 09
            MOV   REMAIN1,AH        ;remainder = AH = 05
;using register indirect addressing mode
            MOV   AL,DATA7          ;AL holds numerator
            SUB   AH,AH             ;AH must be cleared
            MOV   BX,OFFSET DATA8   ;BX holds offset of DATA8
            DIV   BYTE PTR [BX]     ;divide AX by DATA8
            MOV   QOUT2,AX
            MOV   REMAIND2,DX
```

word/word: In this case, the numerator is in AX and DX must be cleared. The denominator can be in a register or memory. After the DIV; AX will have the quotient and the remainder will be in DX.

```
MOV   AX,10050      ;AX holds numerator
SUB   DX,DX         ;DX must be cleared
MOV   BX,100        ;BX used for denominator
DIV   BX
MOV   QOUT2,AX      ;quotient = AX = 64H = 100
MOV   REMAIND2,DX   ;remainder = DX = 32H = 50
```

word/byte: Here, the numerator is in AX and the denominator can be in a register or memory. After the DIV instruction, AL will contain the quotient, and AH will contain the remainder. The maximum quotient is FFH.

The following program divides AX = 2055 by CL = 100. Then AL = 14H (20 decimal) is the quotient and AH = 37H (55 decimal) is the remainder.

```
MOV   AX,2055       ;AX holds numerator
MOV   CL,100        ;CL used for denominator
DIV   CL
MOV   QUO,AL        ;AL holds quotient
MOV   REMI,AH       ;AH holds remainder
```

Double-word/word: The numerator is in DX and AX, with the most significant word in DX and the least significant word in AX. The denominator can be in a register or in memory. After the DIV instruction; the quotient will be in AX, and the remainder in DX. The maximum quotient is FFFFH.

```
;from the data segment:
DATA1        DD     105432
DATA2        DW     10000
QUOT         DW     ?
REMAIN       DW     ?
;from the code segment:
             MOV   AX,WORD PTR DATA1        ;AX holds lower word
             MOV   DX,WORD PTR DATA1+2;DX higher word of numerator
             DIV   DATA2
             MOV   QUOT,AX                  ;AX holds quotient
             MOV   REMAIN,DX                ;DX holds remainder
```

- ✓ In the program above, the contents of DX: AX are divided by a word-sized data value, 10000.
- ✓ The 8088/86 automatically uses DX: AX as the numerator anytime the denominator is a word in size.
- ✓ Notice in the example above that DATAl is defined as DD but fetched into a word-size register with the help of WORD PTR. In the absence of WORD PTR, the assembler will generate an error.

Table: Unsigned Division Summary

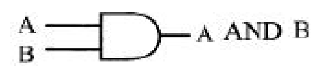
Division	Numerator	Denominator	Quotient	Rem.
byte/byte	AL = byte, AH = 0	register or memory	AL[1]	AH
word/word	AX = word, DX = 0	register or memory	AX[2]	DX
word/byte	AX = word	register or memory	AL[1]	AH
doubleword/word	DXAX = doubleword	register or memory	AX[2]	DX

LOGIC INSTRUCTIONS:

Here, the logic instructions AND, OR, XOR, SHIFT, and COMPARE are discussed with examples.

AND

Inputs		Output
A	**B**	**A AND B**
0	0	0
0	1	0
1	0	0
1	1	1




```
AND destination, source
```

- ✓ This instruction will perform a logical AND on the operands and place the result in the destination. The destination operand can be a register or memory. The source operand can be a register, memory, or immediate.
- ✓ AND will automatically change the CF and OF to zero, and PF, ZF, and SF are set according to the result. The rest of the flags are either undecided or unaffected.

Show the results of the following:

```
MOV   BL,35H
AND   BL,0FH      ;AND BL with 0FH. Place the result in BL.
```

Solution:

```
35H   0 0 1 1 0 1 0 1
0FH   0 0 0 0 1 1 1 1
05H   0 0 0 0 0 1 0 1
```

Flag settings will be: SF = 0, ZF = 0, PF = 1, CF = OF = 0.

- ✓ AND can be used to mask certain bits of the operand. The task of clearing a bit in a binary number is called **masking**. It can also be used to test for a zero operand.

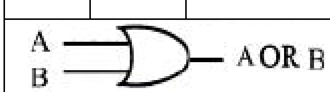
```
  x x x x x x x x   Unknown number
• 0 0 0 0 1 1 1 1   Mask
  0 0 0 0 x x x x   Result
```

```
        AND   DH,DH
        JZ    XXXX
        ...
XXXX:   ...
```

- ✓ The above code will AND DH with itself, and set ZF =1, if the result is zero. This makes the CPU to fetch from the target address XXXX. Otherwise, the instruction below JZ is executed. AND can thus be used to test if a register contains zero.

OR

Inputs		Output
A	**B**	**A OR B**
0	0	0
0	1	1
1	0	1
1	1	1




```
OR    destination,source
```

- ✓ The destination and source operands are ORed and the result is placed in the destination.
- ✓ The destination operand can be a register or in memory. The source operand can be a register, memory, or immediate.
- ✓ OR will automatically change the CF and OF to zero, and PF, ZF,

and SF are set according to the result. The rest of the flags are either undecided or unaffected.

Show the results of the following:

```
        MOV  AX,0504        ;AX = 0504
        OR   AX,0DA68H      ;AX = DF6C
```

Solution:

```
 0504H   0000 0101 0000 0100
 DA68H   1101 1010 0110 1000   Flags will be: SF = 1 , ZF = 0, PF = 1, CF = OF = 0.
 DF6C    1101 1111 0110 1100   Notice that parity is checked for the lower 8 bits only.
```

- ✓ The OR instruction can be used to test for a zero operand. For example, "*OR BL, 0*"will OR the register BL with 0 and make ZF = 1, if BL is zero. "*OR BL, BL*" will achieve the same result.
- ✓ OR can also be used to set certain bits of an operand to 1.

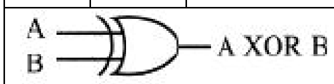
```
    x x x x  x x x x   Unknown number
  + 0 0 0 0  1 1 1 1   Mask
  ------------------
    x x x x  1 1 1 1   Result
```

XOR

```
XOR   dest,src
```

- ✓ The XOR instruction will eXclusive-OR the operands and place the result in the destination. XOR sets the result bits to 1 if they are not equal; otherwise, they are reset to 0.
- ✓ The destination operand can be a register or in memory. The source operand can be a register, memory, or immediate.
- ✓ OR will automatically change the CF and OF to zero, and PF, ZF, and SF are set according to the result. The rest of the flags are either undecided or unaffected.

Inputs		**Output**
A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0



Show the results of the following:

```
        MOV     DH,54H
        XOR     DH,78H
```

Solution:

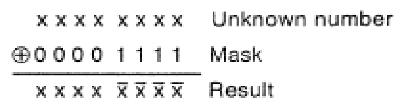
```
54H     0 1 0 1 0 1 0 0
78H     0 1 1 1 1 0 0 0
2C      0 0 1 0 1 1 0 0    Flag settings will be: SF = 0, ZF = 0, PF = 0, CF = OF = 0.
```

The XOR instruction can be used to clear the contents of a register by XORing it with itself. Show how "XOR AH,AH" clears AH, assuming that AH = 45H.

Solution:

```
45H     01000101
45H     01000101
00      00000000      Flag settings will be: SF = 0, ZF = 1, PF =1 , CF = OF = 0.
```

- ✓ XOR can be used to see if two registers have the same value. "*XOR BX, CX*" will make ZF = 1, if both registers have the same value, and if they do, the result (0000) is saved in BX, the destination.
- ✓ XOR can also be used to toggle (invert/compliment) bits of an operand. For example, to toggle bit 2 of register AL:




```
XOR   AL,04H        ;XOR  AL with 0000 0100
```

- ✓ This would cause bit 2 of AL to change to the opposite value; all other bits would remain unchanged.

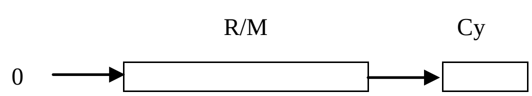
SHIFT

- o Shift instructions shift the contents of a register or memory location right or left.
- o The number of times (or bits) that the operand is shifted can be specified directly if it is once only, or through the CL register if it is more than once.
- o There are two kinds of shifts:
  - ✓ Logical – for unsigned operands
  - ✓ Arithmetic – signed operands.

SHR: This is the logical shift right. The operand is shifted right bit by bit, and for every shift the LSB (least significant bit) will go to the carry flag (CF) and the MSB (most significant bit) is filled with 0.

- ✓ SHR does affect the OF, SF, PF, and ZF flags.
- ✓ The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions. For example, "*SHR 25, CL*" will cause the assembler to give an error.

Eg:
SHR BH, CL



Shift right

	Before	*After*
BH	0100 0100	0001 0001
CL	02H	
Cy	1	0

Show the result of SHR in the following:

```
        MOV    AL,9AH
        MOV    CL,3     ;set number of times to shift
        SHR    AL,CL
```

Solution:

```
        9AH =        10011010
                     01001101      CF = 0 (shifted once)
                     00100110      CF = 1 (shifted twice)
                     00010011      CF = 0 (shifted three times)
```

After shifting right three times, AL = 13H and CF = 0.

- ✓ If the operand is to be shifted once only, this is specified in the SHR instruction itself rather than placing 1 in the CL. This saves coding of one instruction:

```
MOV     BX,0FFFFH     ;BX=FFFFH
SHR     BX,1          ;shift right BX once only
```

- ✓ After the above shift, BX = 7FFFH and CF = 1.

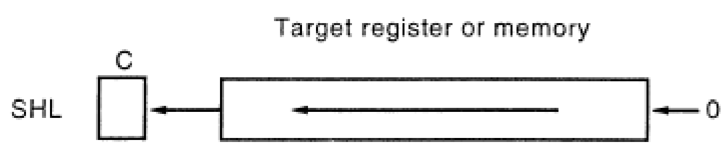
Show the results of SHR in the following:

```
        ;from the data segment:
        DATA1            DW      7777H
        ;from the code segment:
        TIMES            EQU     4
                         MOV     CL,TIMES      ;CL=04
                         SHR     DATA1,CL      ;shift DATA1 CL times
```

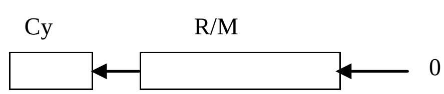
Solution:
After the four shifts, the word at memory location DATA1 will contain 0777. The four LSBs are lost through the carry, one by one, and 0s fill the four MSBs.

SHL: Shift left is also a logical shift. It is the reverse of SHR. After every shift the LSB is filled with 0 and the MSB goes to CF.

- ✓ SHL does affect the OF, SF, PF, and ZF flags.
- ✓ The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions. For example, "*SHL 25, CL*" will cause the assembler to give an error.



Eg:
SHL BH, CL



Shift left without Cy

	Before	*After*
BH	0010 0010	1000 1000
CL	02H	
Cy	1	0

Show the effects of SHL in the following:	Can also be coded as
`` MOV DH,6 MOV CL,4 SHL DH,CL ```	

Show the effects of SHL in the following:

```
        MOV     DH,6
        MOV     CL,4
        SHL     DH,CL
```

Solution:

```
                        00000110
        CF=0            00001100        (shifted left once)
        CF=0            00011000
        CF=0            00110000
        CF=0            01100000        (shifted four times)
```

After the four shifts left, the DH register has 60H and CF = 0.

Can also be coded as

```
        MOV     DH,6
        SHL     DH,1
        SHL     DH,1
        SHL     DH,1
        SHL     DH,1
```

COMPARE of Unsigned Numbers:

```
        CMP     destination,source      ;compare dest and src
```

- ✓ The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged.
- ✓ The destination operand can be in a register or in memory and the source operand can be in a register, memory, or immediate.
- ✓ The compare instruction is really a SUBtraction, except that the values of the operands do not change.
- ✓ The flags are changed according to the execution of SUB. Although all the flags (CF, AF, SF, PF, ZF, and OF flags) are affected, the only ones of interest are ZF and CF.
- ✓ It must be emphasized that in CMP instructions, the operands are unaffected regardless of the result of the comparison. Only the flags are affected.

Table: Flag Settings for Compare Instruction

Compare Operands	CF	ZF	Remark
destination > source	0	0	destination – source; results CF = 0 & ZF = 0
destination = source	0	1	destination – source; results CF = 0 & ZF = 1
destination < source	1	0	destination – source; results CF = 1 & ZF = 0

```
DATA1   DW      235FH
        ...
        MOV     AX,0CCCCH
        CMP     AX,DATA1        ;compare CCCC with 235F
        JNC     OVER            ;jump if CF=0
        SUB     AX,AX
OVER:   INC     DATA1
```

- ✓ In the program above, AX is greater than the contents of memory location DATA1 (0CCCCH > 235FH); therefore, CF = 0 and JNC (jump no carry) will go to target OVER.

```
      MOV   BX,7888H
      MOV   CX,9FFFH
      CMP   BX,CX        ;compare 7888 with 9FFF
      JNC   NEXT
      ADD   BX,4000H
NEXT: ADD   CX,250H
```

- ✓ In the above code, BX is smaller than CX (7888H < 9FFFH), which sets CF = 1, making "*JNC NEXT*" fall through so that "*ADD BX, 4000H*" is executed.
- ✓ In the example above, CX and BX still have their original values (CX = 9FFFH and BX =7888H) after the execution of "*CMP BX, CX*".
- ✓ Notice that CF is always checked for cases of greater or smaller than, but for equal, ZF must be used.

```
TEMP   DB    ?
...
       MOV   AL,TEMP      ;move the TEMP variable into AL
       CMP   AL,99        ;compare AL with 99
       JZ    HOT_HOT      ;if ZF=1 (TEMP = 99) jump to HOT_HOT
       INC   BX           ;otherwise (ZF=0) increment BX
...
HOT_HOT: HLT              ;halt the system
```

- ✓ The above program sample has a variable named TEMP, which is being checked to see if it has reached 99.

In the following Program the CMP instruction is used to search for the highest byte in a series of 5 bytes defined in the data segment.

- ✓ The instruction "*CMP AL, [BX]*" works as follows ([BX] is the contents of the memory location pointed at by register BX).
  - If AL < [BX], then CF = 1 and [BX] becomes the basis of the new comparison.
  - If AL > [BX], then CF = 0 and AL is the larger of the two values and remains the basis of comparison.
- ✓ Although JC (jump carry) and JNC (jump no carry) check the carry flag and can be used after a compare instruction, it is recommended that JA (jump above) and JB (jump below) be used because,
  - The assemblers will unassembled JC as JB, and JNC as JA.
- ✓ The below Program searches through five data items to find the highest grade.
- ✓ The program has a variable called "Highest" that holds the highest grade found so far. One by one, the grades are compared to Highest. If any of them is higher, that value is placed in Highest.
- ✓ This continues until all data items are checked. A REPEAT-UNTIL structure was chosen in the program design.
- ✓ The program uses register AL to hold the highest grade found so far. AL is given the initial value of 0. A loop is used to compare each of the 5 bytes with the value in AL.

- ✓ If AL contains a higher value, the loop continues to check the next byte. If AL is smaller than the byte being checked, the contents of AL are replaced by that byte and the loop continues.

Assume that there is a class of five people with the following grades: 69, 87, 96, 45, and 75. Find the highest grade.

```
TITLE       PROG3-3 (EXE) CMP EXAMPLE
PAGE        60,132
.MODEL SMALL
.STACK 64
;------------------
            .DATA
GRADES      DB    69,87,96,45,75
            ORG   0008
HIGHEST     DB    ?
;------------------
            .CODE
MAIN        PROC  FAR
            MOV   AX,@DATA
            MOV   DS,AX
            MOV   CX,5                 ;set up loop counter
            MOV   BX,OFFSET GRADES     ;BX points to GRADE data
            SUB   AL,AL          ;AL holds highest grade found so far
AGAIN:      CMP   AL,[BX]              ;compare next grade to highest
            JA    NEXT                 ;jump if AL still highest
            MOV   AL,[BX]              ;else AL holds new highest
NEXT:       INC   BX                   ;point to next grade
            LOOP  AGAIN                ;continue search
            MOV   HIGHEST,AL           ;store highest grade
            MOV   AH,4CH
            INT   21H                  ;go back to OS
MAIN        ENDP
            END   MAIN
```

Program 3-3

NOTE:

There is a relationship between the pattem of lowercase and uppercase letters, as shown below for *A* and *a*:

A	*0100 0001*	*41H*
a	*0110 0001*	*61H*

The only bit that changes is d5. To change from lowercase to uppercase , d5 must be masked.

Note that small and capital letters in ASCII have the following values:

Letter	Hex	Binary	Letter	Hex	Binary
A	41	0100 0001	a	61	0110 0001
B	42	0100 0010	b	62	0110 0010
C	43	0100 0011	c	63	0110 0011
...	...	...	...	...	...
Y	59	0101 1001	y	79	0111 1001
Z	5A	0101 1010	z	7A	0111 1010

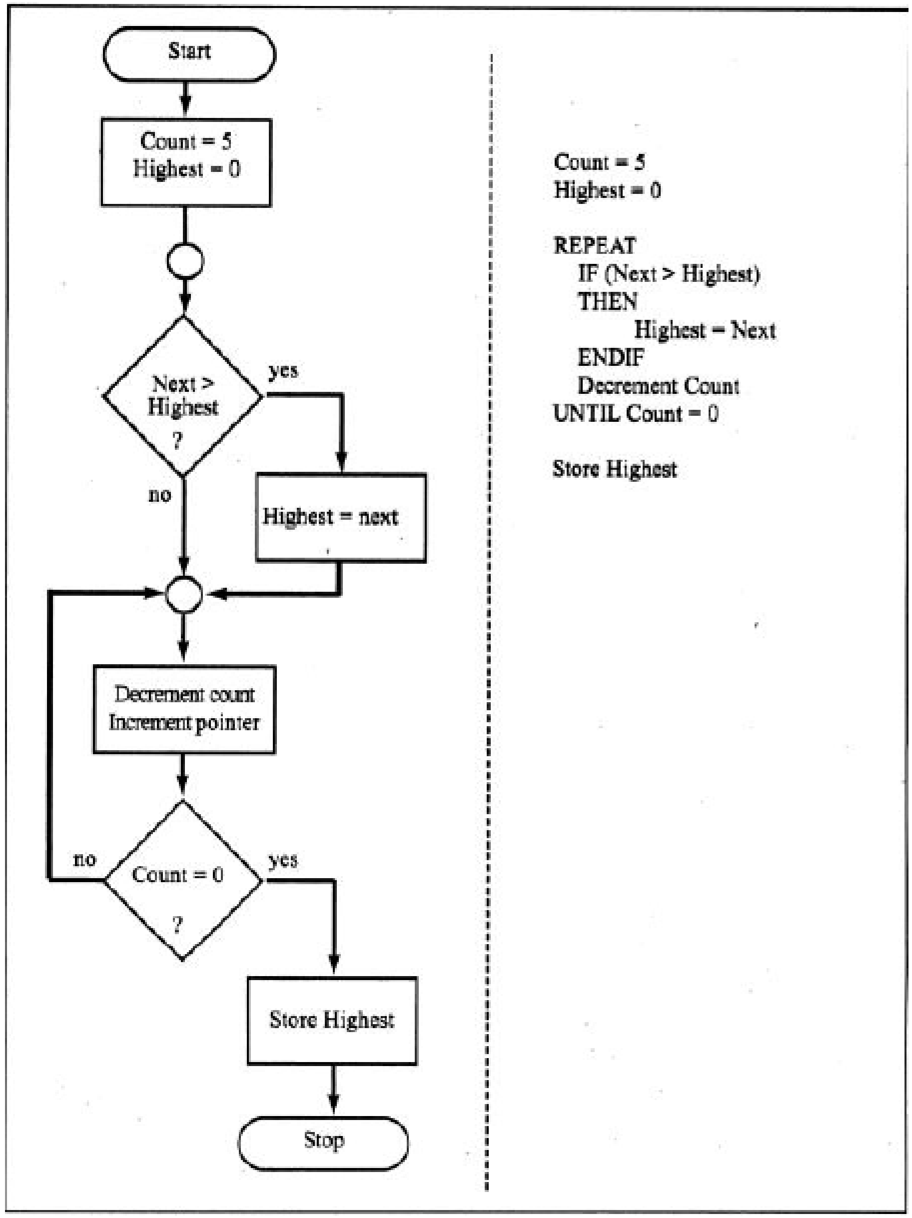


Fig: Flowchart and Pseudocode for Program 3-3

The following Program uses the CMP instruction to determine if an ASCII character is uppercase or lowercase.

- ✓ The following Program first detects if the letter is in lowercase, and if it is, it is ANDed wit h 1101 1111B = DFH. Otherwise, it is simply left alone.
- ✓ To determine if it is a lowercase letter, it is compared with 61H and 7AH to see if it is in the range a to z. Anything above or below this range should be left alone.

In the following Program, 20H could have been subtracted from the lowercase letters instead of ANDing with 1101 1111B.

```
TITLE       PROG3-4 (EXE) LOWERCASE TO UPPERCASE CONVERSION
PAGE        60,132
.MODEL SMALL
.STACK 64
;----------------
            .DATA
DATA1       DB     'mY NAME is jOe'
            ORG    0020H
DATA2       DB     14 DUP(?)
;----------------
      .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    SI,OFFSET DATA1   ;SI points to original data
     MOV    BX,OFFSET DATA2   ;BX points to uppercase data
     MOV    CX,14             ;CX is loop counter
BACK:MOV    AL,[SI]           ;get next character
     CMP    AL,61H            ;if less than 'a'
     JB     OVER              ;then no need to convert
     CMP    AL,7AH            ;if greater than 'z'
     JA     OVER              ;then no need to convert
     AND    AL,11011111B      ;mask d5 to convert to uppercase
OVER:MOV    [BX],AL           ;store uppercase character
     INC    SI                ;increment pointer to original
     INC    BX                ;increment pointer to uppercase data
     LOOP   BACK              ;continue looping if CX > 0
     MOV    AH,4CH
     INT    21H               ;go back to OS
MAIN ENDP
     END    MAIN
```

Program 3-4

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD AND ASCII CONVERSION:

- BCD (*binary coded decimal*) is needed because we use the digits 0 to 9 for numbers in everyday life. Binary representation of 0 to 9 is called BCD.
- In computer literature, one encounters two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD.

Unpacked BCD:

- In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.
  - Example: "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively.
- In the case of unpacked BCD it takes 1 byte of memory location or a register of 8 bits to contain the number.

Packed BCD:

- In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
    - For example, "0101 1001" is packed BCD for 59.
- It takes only 1 byte of memory to store the packed BCD operands. This is one reason to use packed BCD since it is twice as efficient in storing data.

ASCII Numbers:

- In ASCII keyboards, when key "0" is activated, for example, "011 0000" (30H) is provided to the computer. In the same way, 31H (011 0001) is provided for key "1", and so on, as shown in the following list:

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

It must be noted that, although ASCII is standard in many countries, BCD numbers have universal application. So, the data conversion from ASCII to BCD and vice versa should be studied.

ASCII to BCD Conversion:

To process data in BCD, first the ASCII data provided by the keyboard must be converted to BCD. Whether it should be converted to packed or unpacked BCD depends on the instructions to be used.

ASCII to Unpacked BCD Conversion:

To convert ASCII data to BCD, the programmer must get rid of the tagged "011" in the higher 4 bits of the ASCII. To do that, each ASCII number is ANDed with "0000 1111" (0FH), as shown in the next example. These programs show three different methods for converting the 10 ASCII digits to unpacked BCD. All use the same data segment:

```
ASC       DB      '9562481273'
          ORG     0010H
UNPACK    DB      10 DUP(?)
```

The data is defined as DB.

- In the following Program 3-5a; the data is accessed in word-sized chunks.
- The Program 3-5b used the PTR directive to access the data.

- The Program 5-3c uses the based addressing mode (BX+ASC is used as a pointer.

```
        MOV   CX,5
        MOV   BX,OFFSET ASC      ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK   ;DI points to unpacked BCD data
AGAIN:  MOV   AX,[BX]            ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH           ;remove ASCII 3s
        MOV   [DI],AX            ;store unpacked BCD
        ADD   DI,2               ;point to next unpacked BCD data
        ADD   BX,2               ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5a

```
        MOV   CX,5               ;CX is loop counter
        MOV   BX,OFFSET ASC      ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK   ;DI points to unpacked BCD data
AGAIN:  MOV   AX,WORD PTR [BX]   ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH           ;remove ASCII 3s
        MOV   WORD PTR [DI],AX   ;store unpacked BCD
        ADD   DI,2               ;point to next unpacked BCD data
        ADD   BX,2               ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5b

```
        MOV   CX,10              ;load the counter
        SUB   BX,BX              ;clear BX
AGAIN:  MOV   AL,ASC[BX]         ;move to AL content of mem [BX+ASC]
        AND   AL,0FH             ;mask the upper nibble
        MOV   UNPACK[BX],AL      ;move to mem [BX+UNPACK] the AL
        INC   BX                 ;point to next byte
        LOOP  AGAIN              ;loop until it is finished
```

Program 3-5c

ASCII to Packed BCD Conversion:

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of the 3) and then combined to make packed BCD.

For example, for 9 and 5 the keyboard gives 39 and 35, respectively. The goal is to produce 95H or"1001 0101", which is called packed BCD. This process is illustrated in detail below:

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

```
            ORG   0010H
VAL_ASC     DB    '47'
VAL_BCD     DB    ?
;reminder:  DB will put 34 in 0010H location and 37 in 0011H
            MOV   AX,WORD PTR VAL_ASC     ;AH=37,AL=34
            AND   AX,0F0FH                ;mask 3 to get unpacked BCD
            XCHG  AH,AL                   ;swap AH and AL.
            MOV   CL,4                    ;CL=04 to shift 4 times
            SHL   AH,CL                   ;shift left AH to get AH=40H
            OR    AL,AH                   ;OR them to get packed BCD
            MOV   VAL_BCD,AL              ;save the result
```

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format. There are special instructions, such as DAA and DAS, which require that the data be in packed BCD form and give the result in packed BCD.

- For the result to be displayed on the monitor or be printed by the printer, it must be in ASCII format. Conversion from packed BCD to ASCII is discussed next.

Packed BCD to ASCII Conversion:

To convert packed BCD to ASCII, it must first be converted to unpacked and then the unpacked BCD is tagged with 011 0000 (30H).

The following shows the process of converting from packed BCD to ASCII:

```
Packed BCD   Unpacked BCD               ASCII
29H          02H        & 09H           32H      & 39H
0010 1001    0000 0010 & 0000 1001      011 0010 & 011 1001

VAL1_BCD     DB     29H
VAL3-ASC     DW     ?
             ...
             MOV    AL,VAL1_BCD
             MOV    AH,AL         ;copy AL to AH. now AH=29,AL=29H
             AND    AX,0F00FH     ;mask 9 from AH and 2 from AL
             MOV    CL,4          ;CL=04 for shift
             SHR    AH,CL         ;shift right AH to get unpacked BCD
             OR     AX,3030H      ;combine with 30 to get ASCII
             XCHG   AH,AL         ;swap for ASCII storage convention
             MOV    VAL3_ASC,AX   ;store the ASCII
```

- After learning bow to convert ASCII to BCD, the application of BCD numbers is the next step.
- There are two instructions that deal specifically with BCD numbers: DAA and DAS.

BCD Addition and Correction:

In BCD addition, after adding packed BCD numbers, the result is no longer BCD. Look at this example:

```
MOV AL,17H
ADD AL,28H
```

Adding them gives 0011 1111B (3FH), which is not BCD! A BCD number can- only have digits from 0000 to 1001 (or 0 to 9). The result above should have been 17+ 28 = 45 (0100 0101).

✓ To correct this problem, the programmer must add 6 (0110) to the low digit: 3F + 06 = 45H.

The same problem could have happened in the upper digit (for example, in 52H + 87H = D9H).

✓ Again to solve this problem, 6 must be added to the upper digit (D9H + 60H = 139H), to ensure that the result is BCD (52 + 87 = 139).

DAA

The DAA (*decimal adjust for addition*) instruction in x86 microprocessors is provided exactly for the purpose of correcting the problem associated with BCD addition. DAA will add 6 to the lower nibble or higher nibble if needed; otherwise, it will leave the result alone.

The following example will clarify these points:

```
DATA1    DB  47H
DATA2    DB  25H
DATA3    DB?
         MOV AL,DATA1        ;AL holds first BCD operand
         MOV BL,DATA2        ;BL holds second BCD operand
         ADD AL,BL           ;BCD addition
         DAA                 ;adjust for BCD addition
         MOV DATA3,AL        ;store result in correct BCD form
```

After the program is executed, the DATA3 field will contain 72H (47 + 25 =72).

- ✓ Note that DAA works only on AL. In other words, while the source can be an operand of any addressing mode, the destination must be AL in order for DAA to work.
- ✓ It needs to be emphasized that DAA must be used after the addition of BCD operands and that BCD operands can never have any digit greater than 9. In other words, no A-F digit is allowed.
- ✓ It is also important to note that DAA works only after an ADD instruction; it will not work after the INC instruction.

Summary of DAA Action:

1. If after an ADD or ADC instruction the lower nibble (4 bits) is greater than 9, or if AF = 1, add 0110 to the lower 4 bits.
2. If the upper nibble is greater than 9, or if CF = 1, add 0110 to the upper nibble.

In reality there is no other use for the AF (auxiliary flag) except for BCD addition and correction. For example, adding 29H and 18H will result in 41H, which is incorrect as far as BCD is concerned.

See the following code:

```
  Hex    BCD
  29     0010 1001
+ 18   + 0001 1000
  41     0100 0001
+  6   +      0110
  47     0100 0111
```

Because AF = 1,
DAA adds 6 to lower nibble.
The final result is BCD.

```
  Hex    BCD
  53     0010 0011
+ 75   + 0111 0101
  D8     1101 1000
+ 6    +  0110
 128     0010 1000
```

Because the upper nibble is greater than 9,
DAA adds 6 to upper nibble.
The final result is BCD.

The above example shows that 6 is added to the upper nibble due to the fact it is greater than 9.

Eg1:

```
                    ; AL = 0011 1001 = 39 BCD
                    ; CL = 0001 0010 = 12 BCD
ADD AL, CL          ; AL = 0100 1011 = 4BH
DAA                 ; Since 1011 > 9; Add correction factor 06.
                    ; AL = 0101 0001 = 51 BCD
```

Eg2: ; AL = 1001 0110 = 96 BCD

```
                          ; BL = 0000 0111 = 07 BCD
ADD AL, BL                ; AL = 1001 1101 = 9DH
DAA                       ; Since 1101 > 9; Add correction factor 06
                          ; AL = 1010 0011 = A3H
                          ; Since 1010 > 9; Add correction factor 60
                          ; AL = 0000 0011 = 03 BCD. The result is 103.
```

More Examples:

1: Add decimal numbers 22 and 18.

```
MOV AL, 22H      ; (AL)= 22H
ADD AL, 18H      ; (AL) = 3AH Illegal, incorrect answer!
DAA              ; (AL) = 40H   Just treat it as decimalwith CF = 0
```

```
 3AH             In this case, DAA same as ADD AL, 06H
+06H             When LS hex digit in AL is >9, add 6 to it
=40H
```

2: Add decimal numbers 93 and 34.

```
MOV AL, 93H      ; (AL)= 93H
ADD AL, 34H      ; (AL) = C7H, CF = 0   Illegal & Incorrect!
DAA              ; (AL) = 27H   Just treat it as decimal with CF = 1
```

```
 C7H             In this case, DAA same as ADD AL, 60H
+60H             When MS hex digit in AL is >9, add 6 to it
=27H
```

3: Add decimal numbers 93 and 84.

```
MOV AL, 93H      ; (AL)= 93H
ADD AL, 84H      ; (AL) = 17H, CF = 1   Incorrect answer!

DAA              ; (AL) = 77H   Just treat it as decimal with CF = 1 (carry generated?)
```

```
 17H             In this case, DAA same as ADD AL, 60H
+60H             When CF = 1, add 6 to MS hex digit of AL and treat
=77H             Carry as 1 even though not generated in this addition
```

4: Add decimal numbers 65 and 57.

```
MOV AL, 65H      ; (AL)= 65H
ADD AL, 57H      ; (AL) = BCH
DAA              ; (AL) = 22H   Just treat it as decimal with CF = 1
```

```
 BCH             In this case, DAA same as ADD AL, 66H
+66H
=22H  CF = 1
```

5: Add decimal numbers 99 and 28.

```
MOV AL, 99H      ; (AL)= 99H
ADD AL, 28H      ; (AL) = C1H, AF = 1
DAA              ; (AL) = 27H   Just treat it as decimal with CF = 1
```

```
 C1H             In this case, DAA same as ADD AL, 66H
+66H             6 added to LS hex digit of AL, as AF = 1
=27H  CF = 1     6 added to MS hex digit of AL, as it is >9
```

6: Add decimal numbers 36 and 42.

```
MOV AL, 36H        ; (AL)= 36H
ADD AL, 42H        ; (AL) = 78H
DAA                ; (AL) = 78H   Just treat it as decimal with CF = 0

  78H
+00H               In this case, DAA same as ADD AL, 00H
=78H
```

The following Program demonstrates the use of DAA after addition of multibyte packed BCD numbers.

Two sets of ASCII data have come in from the keyboard. Write and run a program to:
1. Convert from ASCII to packed BCD.
2. Add the multibyte packed BCD and save it.
3. Convert the packed BCD result to ASCII.

```
TITLE       PROG3-6 (EXE) ASCII TO BCD CONVERSION AND ADDITION
PAGE        60,132
.MODE SMALL
.STACK 64
;---------------------
            .DATA
DATA1_ASC   DB     '0649147816'
            ORG    0010H
DATA2_ASC   DB     '0072687188'
            ORG    0020H
DATA3_BCD   DB     5 DUP (?)
            ORG    0028H
DATA4_BCD   DB     5 DUP (?)
            ORG    0030H
DATA5_ADD   DB     5 DUP (?)
            ORG    0040H
DATA6_ASC   DB     10 DUP (?)
;-----------------------
            .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    BX,OFFSET DATA1_ASC     ;BX points to first ASCII data
     MOV    DI,OFFSET DATA3_BCD     ;DI points to first BCD data
     MOV    CX,10                   ;CX holds number bytes to convert
     CALL   CONV_BCD                ;convert ASCII to BCD
     MOV    BX,OFFSET DATA2_ASC     ;BX points to second ASCII data
     MOV    DI,OFFSET DATA4_BCD     ;DI points to second BCD data
     MOV    CX,10                   ;CX holds number bytes to convert
     CALL   CONV_BCD                ;convert ASCII to BCD
     CALL   BCD_ADD                 ;add the BCD operands
     MOV    SI,OFFSET DATA5_ADD     ;SI points to BCD result
     MOV    DI,OFFSET DATA6_ASC     ;DI points to ASCII result
     MOV    CX,05                   ;CX holds count for convert
     CALL   CONV_ASC                ;convert result to ASCII
     MOV    AH,4CH
     INT    21H                     ;go back to OS
MAIN ENDP
;-----------------------
```

```
;THIS SUBROUTINE CONVERTS ASCII TO PACKED BCD
CONV_BCD PROC
AGAIN:      MOV   AX,[BX]       ;BX=pointer for ASCII data
      XCHG  AH,AL
      AND   AX,0F0FH     ;mask ASCII 3s
      PUSH  CX           ;save the counter
      MOV   CL,4         ;shift AH left 4 bits
      SHL   AH,CL        ;to get ready for packing
      OR    AL,AH        ;combine to make packed BCD
      MOV   [DI],AL            ;DI=pointer for BCD data
      ADD   BX,2         ;point to next 2 ASCII bytes
      INC   DI           ;point to next BCD data
      POP   CX           ;restore loop counter
      LOOP  AGAIN
      RET
CONV_BCD ENDP
;----------------
;THIS SUBROUTINE ADDS TWO MULTIBYTE PACKED BCD OPERANDS
BCD_ADD PROC
       MOV  BX,OFFSET DATA3_BCD     ;BX=pointer for operand 1
       MOV  DI,OFFSET DATA4_BCD     ;DI=pointer for operand 2
       MOV  SI,OFFSET DATA5_ADD     ;SI=pointer for sum
       MOV  CX,05
       CLC
BACK:  MOV  AL,[BX]+4    ;get next byte of operand 1
       ADC  AL,[DI]+4    ;add next byte of operand 2
       DAA               ;correct for BCD addition
       MOV  [SI]+4,AL    ;save sum
       DEC  BX           ;point to next byte of operand 1
       DEC  DI           ;point to next byte of operand 2
       DEC  SI           ;point to next byte of sum
       LOOP       BACK
       RET
BCD_ADD ENDP
;-----------------
;THIS SUBROUTINE CONVERTS FROM PACKED BCD TO ASCII
CONV_ASC PROC
AGAIN2: MOV AL,[SI]      ;SI=pointer for BCD data
      MOV   AH,AL        ;duplicate to unpack
      AND   AX,0F00FH    ;unpack
      PUSH  CX           ;save counter
      MOV   CL,04        ;shift right 4 bits to unpack
      SHR   AH,CL        ;the upper nibble
      OR    AX,3030H     ;make it ASCII
      XCHG  AH,AL        ;swap for ASCII storage convention
      MOV   [DI],AX            ;store ASCII data
      INC   SI           ;point to next BCD data
      ADD   DI,2         ;point to next ASCII data
      POP   CX           ;restore loop counter
      LOOP  AGAIN2
      RET
CONV_ASC ENDP
      END   MAIN
```

Program 3-6

BCD Subtraction and Correction:

The problem associated with the addition of packed BCD numbers also shows up in subtraction. Again, there is an instruction (DAS) specifically designed to solve the problem.

Therefore, when subtracting packed BCD (single-byte or multibyte) operands, the DAS instruction is put after the SUB or SBB instruction. AL must be used as the destination register to make DAS work.

Summary of DAS Action:

1. If after a SUB or SBB instruction the lower nibble is greater than 9, or if AF = 1 , subtract 0110 from the lower 4 bits.
2. If the upper nibble is greater than 9, or CF = 1, subtract 0110 from the upper nibble.

Due to the widespread use of BCD numbers, a specific data directive, DT, has been created. DT can be used to represent BCD numbers from 0 to $10^{20} - 1$ (that is, twenty 9s).

Assume that the following operands represent the budget, the expenses, and the balance, which is the budget minus the expenses.

```
BUDGET      DT    87965141012
EXPENSES    DT    31610640392
BALANCE     DT    ?             ;balance = budget - expenses

      MOV    CX,10              ;counter=10
      MOV    BX,00              ;pointer=0
      CLC                       ;clear carry for the 1st iteration
      BACK: MOVAL,BYTE PTR BUDGET[ BX]  ;get a byte of the BUDGET
      SBB    AL,BYTE PTR EXPENSES[ BX]  ;subtract a byte from it
      DAS                             ;correct the result for BCD
      MOV    BYTE PTR BALANCE[ BX] ,AL ;save it in BALANCE
      INC    BX                       ;increment for the next byte
      LOOP   BACK                     ;continue until CX=0
```

Notice in the code section above that,

- ✓ no H (hex) indicator is needed for BCD numbers when using the DT directive, and
- ✓ the use of the based relative addressing mode (BX + displacement) allows access to all three arrays with a single register BX.

```
Eg1:                          ; AL = 0011 0010 = 32 BCD
                              ; CL = 0001 0111 = 17 BCD
        SUB AL, CL            ; AL = 0001 1011 = 1BH
        DAS                   ; Subtract 06, since 1011 > 9.
                              ; AL = 0001 0101 = 15 BCD

Eg2:                          ; AL = 0010 0011 = 23 BCD
                              ; CL = 0101 1000 =58 BCD
        SUB AL, CL            ; AL = 1100 1011 = CBH
        DAS                   ; Subtract 66, since 1100 >9 & 1011 > 9.
                              ; AL = 0110 0101 = 65 BCD, CF = 1.
                ; Since CF = 1, answer is – 65.
```

More Examples:

1: Subtract decimal numbers 45 and 38.

```
MOV AL, 45H        ; (AL)= 45H
SUB AL, 38H        ; (AL) = 0DH   Illegal, incorrect answer!
 DAS               ; (AL) = 07H   Just treat it as decimal  with Cy = 0

  0DH              In this case, DAS same as SUB AL, 06H
-06H               When LS hex digit in AL is >9, subtract 6
=07H
```

2: Subtract decimal numbers 63 and 88.

```
MOV AL, 63H        ; (AL)= 63H
SUB AL, 88H        ; (AL) = DBH, Cy=1   Illegal & Incorrect!
DAS                ; (AL) = 75H   Just treat it as decimal with Cy = 1 (carry generated?)

  DBH              In this case, DAS same as SUB AL, 66H
-66H               When Cy = 1, it means result is negative
=75H               Result is 75, which is 10's complement of 25
                   Treat Cy as 1 as Cy was generated in the previous subtraction itself!
```

3: Subtract decimal numbers 45 and 52.

```
MOV AL, 45H        ; (AL)= 45H
SUB AL, 52H        ; (AL) = F3H, Cy = 1   Incorrect answer!
DAS                ; (AL) = 93H   Just treat it as decimal with Cy = 1 (carry generated?)

  F3H              In this case, DAS same as SUB AL, 60H
-60H               When Cy = 1, it means result is negative
=93H               Result is 93, which is 10's complement of 07
```

4: Subtract decimal numbers 50 and 19.

```
MOV AL, 50H        ; (AL)= 50H
SUB AL, 19H        ; (AL) = 37H, Ac = 1
DAS                ; (AL) = 31H   Just treat it as decimal with Cy =0

  37H              In this case, DAS same as SUB AL, 06H
-06H               06H is subtracted from AL as Ac = 1
=31H
```

5: Subtract decimal numbers 99 and 88.

```
MOV AL, 99H        ; (AL)= 99H
SUB AL, 88H        ; (AL) = 11H
DAS                ; (AL) = 11H   Just treat it as decimal with Cy = 0

  11H              In this case, DAS same as SUB AL, 00H
-00H
=11H
```

6: Subtract decimal numbers 14 and 92.

```
MOV AL, 14H        ; (AL)= 14H
SUB AL, 92H        ; (AL) = 82H, Cy = 1
DAS                ; (AL) = 22H   Just treat it as decimal with Cy = 1

  82H              In this case, DAS same as SUB AL, 60H
-60H               60H is subtracted from AL as Cy = 1
=22H               22 is 10's complement of 78
```

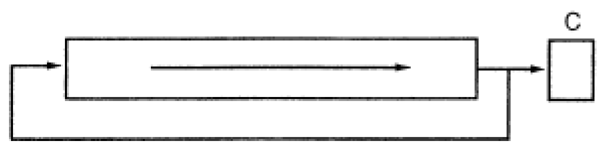
ROTATE INSTRUCTIONS:

In many applications there is a need to perform a bitwise rotation of an operand. The rotation instructions ROR, ROL and RCR, RCL are designed specifically for that purpose. They allow a program to rotate an operand right or left.

- In rotate instructions, the operand can be in a register or memory. If the number of times an operand is to be rotated is more than 1, this is indicated by CL. This is similar to the shift instructions.
- There are two types of rotations. One is a simple rotation of the bits of the operand, and the other is a rotation through the carry.

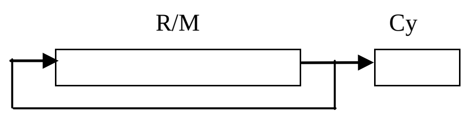
ROR (rotate right)

In rotate right, as bits are shifted from left to right they exit from the right end (LSB) and enter the left end (MSB). In addition, as each bit exits the LSB, a copy of it is given to the carry flag. In other words, in ROR, the LSB is moved to the MSB and is also copied to CF, as shown in the diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL is used to hold the number of times it is to be rotated.

Eg:
ROR BH, 1



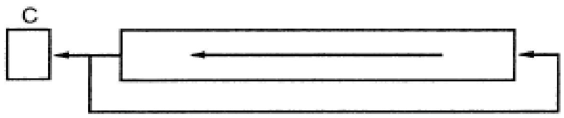
Rotate right without Cy

	Before	*After*
BH	0100 0010	0010 0001
Cy	1	0

```
        MOV     AL,36H          ;AL=0011 0110
        ROR     AL,1            ;AL=0001 1011  CF=0
        ROR     AL,1            ;AL=1000 1101  CF=1
        ROR     AL,1            ;AL=1100 0110  CF=1
;or:
        MOV     AL,36H          ;AL=0011 0110
        MOV     CL,3            ;CL=3 number of times to rotate
        ROR     AL,CL           ;AL=1100 0110 CF=1
;the operand can be a word:
        MOV     BX,0C7E5H       ;BX=1100 0111 1110 0101
        MOV     CL,6            ;CL=6 number of times to rotate
        ROR     BX,CL           ;BX=1001 0111 0001 1111 CF=1
```

ROL (rotate left)

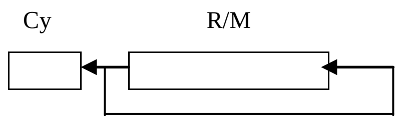
In rotate left, as bits are shifted from right to left they exit the left end (MSB) and enter the right end (LSB). In addition, every bit that leaves the MSB is copied to the carry flag. In other words, in ROL the MSB is moved to the LSB and is also copied to CF, as shown in the diagram.



If the operand is to be rotated once, the 1 is coded. Otherwise, the number of times it is to be rotated is in CL.

Eg:

ROL BH, CL



Rotate left without Cy

	Before	*After*
BH	0010 0010	1000 1000
CL	02H	
Cy	1	0

```
        MOV   BH,72H     ;BH=0111 0010
        ROL   BH,1       ;BH=1110 0100  CF=0
        ROL   BH,1       ;BH=1100 1001  CF=1
        ROL   BH,1       ;BH=1001 0011  CF=1
        ROL   BH,1       ;BH=0010 0111  CF=1
;or:
        MOV   BH,72H     ;BH=0111 0010
        MOV   CL,4       ;CL=4 number of times to rotate
        ROL   BH,CL      ;BH=0010 0111  CF=1

;The operand can be a word:
        MOV   DX,672AH   ;DX=0110 0111 0010 1010
        MOV   CL,3       ;CL=3 number of times to rotate
        ROL   DX,CL ;DX=0011 1001 0101 0011 CF=1
```

The following Program shows an application of the rotation instruction. The maximum count in Program will be 8 since the program is counting the number of 1s in a byte of data. If the operand is a 16-bit word, the number of 1s can go as high as 16.

Write a program that finds the number of 1s in a byte.

```
;From the data segment:
DATA1       DB          97H
COUNT       DB          ?
;From the code segment:
            SUB   BL,BL        ;clear BL to keep the number of 1s
            MOV   DL,8         ;rotate total of 8 times
            MOV   AL,DATA1
AGAIN:      ROL   AL,1         ;rotate it once
            JNC   NEXT         ;check for 1
            INC   BL           ;if CF=1 then add one to count
NEXT:       DEC   DL           ;go through this 8 times
            JNZ   AGAIN        ;if not finished go back
            MOV   COUNT,BL     ;save the number of 1s
```

Program 3-7

The Program is similar to the previous one, rewritten for a word-sized operand. It also provides the count in BCD format instead of hex. Reminder: AL is used to make a BCD counter because the because, the DAA instruction works only on AL.

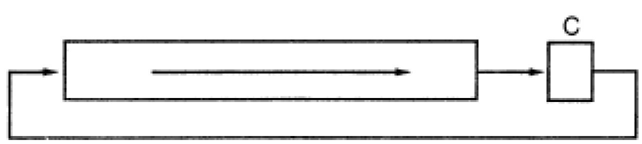
Write a program to count the number of 1s in a word. Provide the count in BCD.

```
DATAW1     DW          97F4H
COUNT2     DB          ?
           ...
           SUB    AL,AL       ;clear AL to keep the number of 1s in BCD
           MOV    DL,16       ;rotate total of 16 times
           MOV    BX,DATAW1   ;move the operand to BX
AGAIN:     ROL    BX,1        ;rotate it once
           JNC    NEXT        ;check for 1. If CF=0 then jump
           ADD    AL,1        ;if CF=1 then add one to count
           DAA                ;adjust the count for BCD
NEXT:      DEC    DL          ;go through this 16 times
           JNZ    AGAIN       ;if not finished go back
           MOV    COUNT2,AL   ;save the number of 1s in COUNT2
```

Program 3-8

RCR (rotate right through carry)

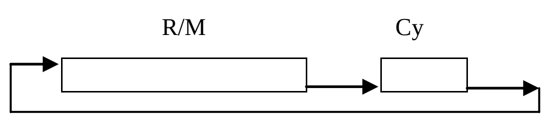
In RCR, as bits are shifted from left to right, they exit the right end (LSB) to the carry flag, and the carry flag enters the left end (MSB). In other words, in RCR the LSB is moved to CF and CF is moved to the MSB. In reality, CF acts as if it is part of the operand. This is shown in the diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, the register CL holds the number of times.

Eg:

RCR BH, 1



Rotate right with Cy

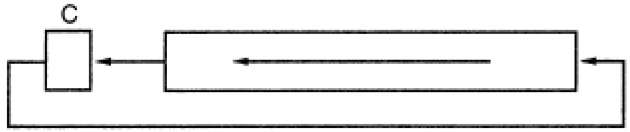
	Before	*After*
BH	0100 0010	1010 0001
Cy	1	0

```
        CLC                         ;make CF=0
        MOV    AL,26H               ;AL=0010 0110
        RCR    AL,1                 ;AL=0001 0011 CF=0
        RCR    AL,1                 ;AL=0000 1001 CF=1
        RCR    AL,1                 ;AL=1000 0100 CF=1
or:
        CLC                         ;make CF=0
        MOV    AL,26H               ;AL=0010 0110
        MOV    CL,3                 ;CL=3 number of times to rotate
        RCR    AL,CL                ;AL=1000 0100 CF=1

;the operand can be a word
        STC                         ;make CF=1
        MOV    BX,37F1H             ;BX=0011 0111 1111 0001
        MOV    CL,5                 ;CL=5 number of times to rotate
        RCR    BX,CL                ;BX=0001 1001 1011 1111 CF=0
```

RCL (rotate left through carry)

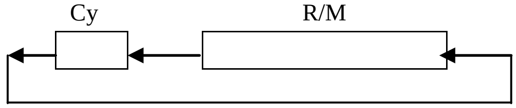
In RCL, as bits are shifted from right to left, they exit the left end (MSB) and enter the carry flag, and the carry flag enters the right end (LSB). In other words, in RCL the MSB is moved to CF and CF is moved to the LSB. In reality, CF acts as if it is part of the operand. This is shown in the following diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL holds the number of times.

Eg:

RCL BH, CL



Rotate left with Cy

	Before	*After*
BH	0010 0010	1000 1010
CL	02H	
Cy	1	0

```
        STC                         ;make CF=1
        MOV    BL,15H               ;BL=0001 0101
        RCL    BL,1                 ;0010 1011 CF=0
        RCL    BL,1                 ;0101 0110 CF=0
or:
        STC                         ;make CF=1
        MOV    BL,15H               ;BL=0001 0101
        MOV    CL,2                 ;CL=2 number of times for rotation
        RCL    BL,CL                ;BL=0101 0110 CF=0

;the operand can be a word:
        CLC                         ;make CF=0
        MOV    AX,191CH             ;AX=0001 1001 0001 1100
        MOV    CL,5                 ;CL=5 number of times to rotate
        RCL    AX,CL                ;AX=0010 0011 1000 0001 CF=1
```

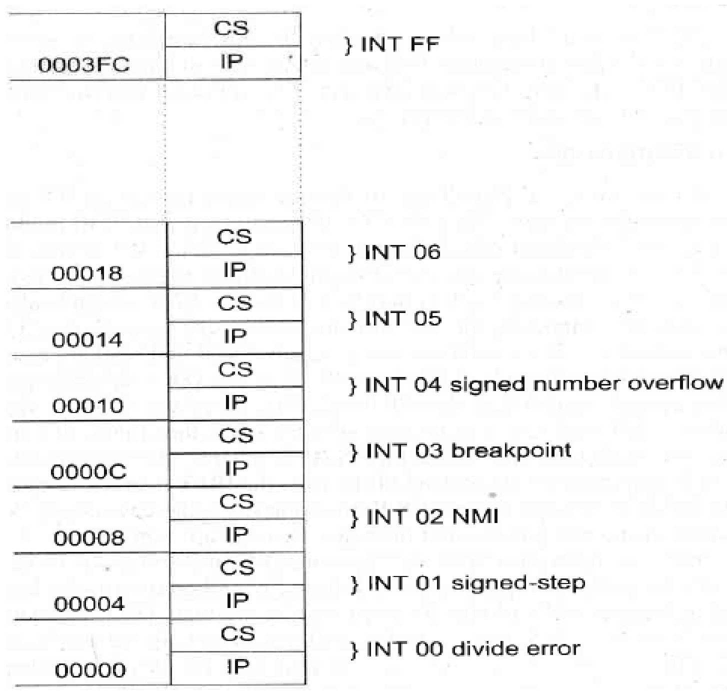
INTERRUPTS IN x86 PC

8088/86 INTERRUPTS

- An interrupt is an external event that informs the CPU that a device needs its service. In 8088/86, there are 256 interrupts: INT 00, INT 01, . . . , INT FF (sometimes called TYPEs).
- When an interrupt is executed, the microprocessor automatically saves the flag register (FR), the instruction pointer (IP), and the code segment register (CS) on the stack; and goes to a fixed memory location.
- In x86 PCs, the memory locations to which an interrupt goes is always four times the value of the interrupt number. For example, INT 03 will go to address 0000CH (4 * 3 = 12 = 0CH). The following Table is a partial list of the interrupt vector table.

Table: Interrupt Vector

INT Number	Physical Address	Logical Address
INT 00	00000	0000 – 0000
INT 01	00004	0000 – 0004
INT 02	00008	0000 – 0008
INT 03	0000C	0000 – 000C
INT 04	00010	0000 – 0010
INT 05	00014	0000 – 0014
. . .	. . .	. . .
INT FF	003FC	0000 – 03FC



Interrupt Service Routine (ISR):

- ✓ For every interrupt there must be a program associated with it.
- ✓ When an interrupt is invoked, it is asked to run a program to perform a certain service. This program is commonly referred to as an *interrupt service routine* (*ISR*). The interrupt service routine is also called the *interrupt handler*.
- ✓ When an interrupt is invoked, the CPU runs the interrupt service routine. As shown in the above Table, for every interrupt there are allocated four bytes of memory in the interrupt vector table. Two bytes are for the IP and the other two are for the CS of the ISR.
- ✓ These four memory locations provide the addresses of the interrupt service routine for which the interrupt was invoked. Thus the lowest 1024 bytes (256 x 4 = 1024) of memory space are set aside for the interrupt vector table and must not be used for any other function.

Find the physical and logical addresses in the interrupt vector table associated with:
(a) INT 12H (b) INT 8

Solution:

(a) The physical addresses for INT 12H are 00048H–0004BH since (4 × 12H = 48H). That means that the physical memory locations 48H, 49H, 4AH, and 4BH are set aside for the CS and IP of the ISR belonging to INT 12H. The logical address is 0000:0048H–0000:004BH.
(b) For INT 8, we have 8 × 4 = 32 = 20H; therefore, memory addresses 00020H, 00021H, 00022H, and 00023H in the interrupt vector table hold the CS:IP of the INT 8 ISR. The logical address is 0000:0020H–0000:0023H.

Difference between INT and CALL Instructions:

The INT instruction saves the CS: IP of the following instruction and jumps indirectly to the subroutine associated with the interrupt. A CALL FAR instruction also saves the CS: IP and jumps to the desired subroutine (procedure).

The differences can be summarized as follows:

CALL Instruction	INT instruction
1. A CALL FAR instruction can jump to any location within the 1M byte address range of the 8088/86 CPU.	1. INT nn goes to a fixed memory location in the interrupt vector table to get the address of the interrupt service routine.
2. A CALL FAR instruction is used by the programmer in the sequence of instructions in the program.	2. An externally activated hardware interrupt can come-in at any time, requesting the attention of the CPU.
3. A CALL FAR instruction cannot be masked (disabled).	3. INT nn belonging to externally activated hardware interrupts can be masked.
4. A CALL FAR instruction automatically saves only CS: IP of the next instruction on the stack.	4. INT nn saves FR (flag register) in addition to CS: IP of the next instruction.
5. At the end of the subroutine that has been called by the CALL FAR instruction, the RETF (return FAR) is the last instruction. RETF pops CS and IP off the stack.	5. The last instruction in the interrupt service routine (ISR) for INT nn is the instruction IRET (interrupt return). IRET pops off the FR (flag register) in addition to CS and IP.

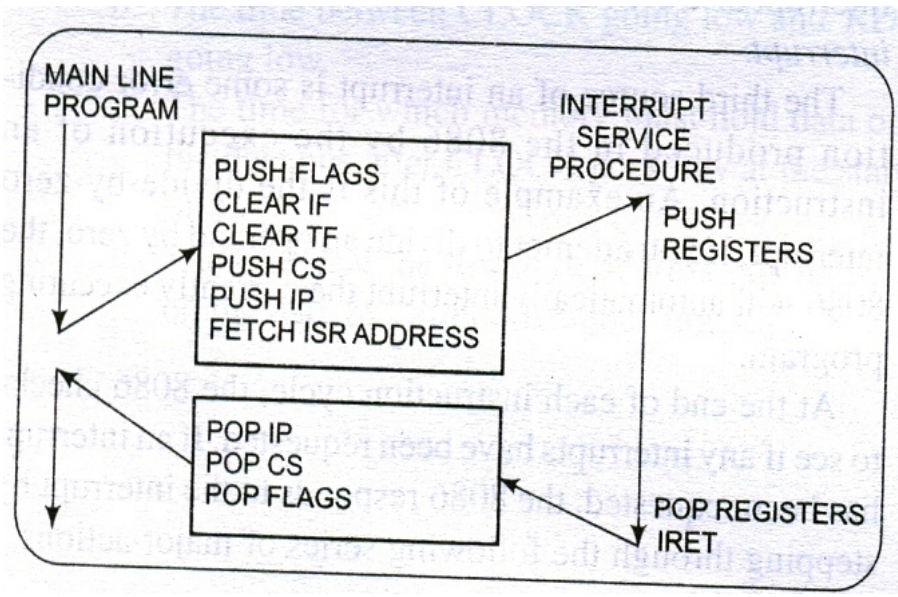
Processing Interrupts:

When the 8088/86 processes any interrupt (software or hardware), it goes through the following steps:

1. The flag register (FR) is pushed onto the stack and SP is decremented by 2, since FR is a 2-byte register.

2. IF (interrupt enable flag) and TF (trap flag) are both cleared (IF = 0 and TF = 0). This masks (causes the system to ignore) interrupt requests from the INTR pin and disables single stepping while the CPU is executing the interrupt service routine.
3. The current CS is pushed onto the stack and SP is decremented by 2.
4. The current IP is pushed onto the stack and SP is decremented by 2.
5. The INT number (type) is multiplied by 4 to get the physical address of the location within the vector table to fetch the CS and IP of the interrupt service routine.
6. From the new CS: IP, the CPU starts to fetch and execute instructions belonging to the ISR program.
7. The last instruction of the interrupt service routine must be IRET, to get IP, CS, and FR back from the stack and make the CPU run the code where it left off.

The following Figure summarizes these steps in diagram form.



Categories of Interrupts:

INT nn is a 2-byte instruction where the first byte is for the opcode and the second byte is the interrupt number. We can have a maximum of 256 (INT 00 INT FFH) interrupts. Of these 256 interrupts, some are used for software interrupts and some are for hardware interrupts.

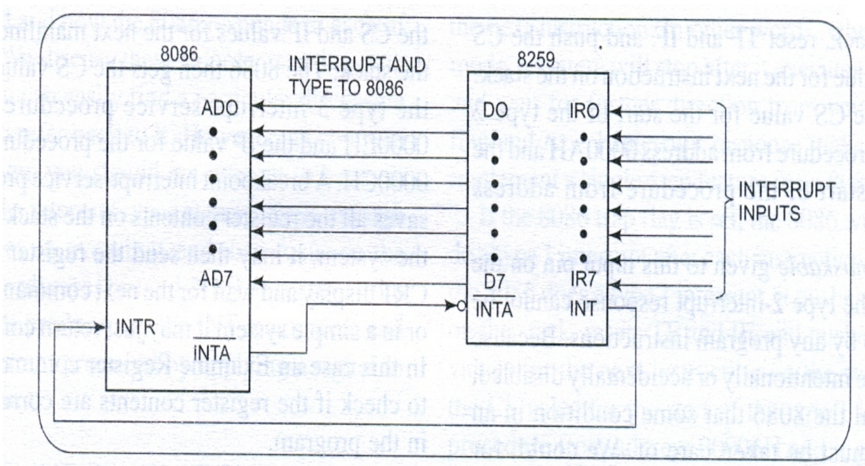
1. **Hardware Interrupts:**

- There are three pins in the x86 that are associated with hardware interrupts. They are INTR (interrupt request), NMI (non-maskable interrupt), and INTA (interrupt acknowledge).
- INTR is an input signal into the CPU, which can be masked (ignored) and unmasked through the use of instructions CLI (clear interrupt flag) and STI (set interrupt flag).

- If IF = 0 (in flag register), all hardware interrupt requests through INTR are ignored. This has no effect on interrupts coming from the NMI pin. The instruction CLI (clear interrupt flag) will make IF = 0.
- To allow interrupt request through the INTR pin, this flag must be set to one (IF = 1). The STI (set interrupt flag) instruction can be used to set IF to 1.
- NMI, which is also an input signal into the CPU, cannot be masked and unmasked using instructions CLI and STI; and for this reason it is called a *non-maskable interrupt*.

- INTR and NMI are activated externally by putting 5V on the pins of NMI and INTR of the x86 microprocessor.
- When either of these interrupts is activated, the x86 finishes the instruction that it is executing, pushes FR and the CS: IP of the next instruction onto the stack, then jumps to a fixed location in the interrupt vector table and fetches the CS: IP for the interrupt service routine (ISR) associated with that interrupt.
- At the end of the ISR, the IRET instruction causes the CPU to get (pop) back its original FR and CS: IP from the stack, thereby forcing the CPU to continue at the instruction where it left off when the interrupt came in.

- Intel has embedded "*INT 02*" into the x86 microprocessor to be used only for NMI.
- Whenever the NMI pin is activated, the CPU will go to memory location 00008 to get the address (CS: IP) of the interrupt service routine (ISR) associated with NMI.
- Memory locations 00008, 00009, 0000A, and 0000B contain the 4 bytes of CS: IP of the ISR belonging to NMI.
- The 8259 programmable interrupt controller (PIC) chip can be connected to INTR to expand the number of hardware interrupts to 64.



2. **Software Interrupts:**

- If an ISR is called upon as a result of the execution of an x86 instruction such as "*INT nn*", it is referred to as software interrupt, since it was invoked from software, not from external hardware.
- Examples of such interrupts are DOS "*INT 21H*" function calls and video interrupts "*INT 10H*".
- These interrupts can be invoked in the sequence of code just like any other x86 instruction.
- Many of the interrupts in this category are used by the MS DOS operating system and IBM BIOS to perform essential tasks that every computer must provide to the system and the user.
- Within this group of interrupts there are also some *predefined functions* associated with some of the interrupts. They are "*INT 00*" (divide error), "*INT 01*" (single step), "*INT 03*" (breakpoint), and "*INT 04*" (signed number overflow). Each is described below.
- The rest of the interrupts from "*INT 05*" to "*INT FF*" can be used to implement either software or hardware interrupts.

Functions associated with INT 00 to INT 04:

Interrupts INT 00 to INT 04 have predefined tasks (functions) and cannot be used in any other way.

INT 00 (divide error)

- ✓ This interrupt belongs to the category of interrupts referred to as conditional or exception interrupts. Internally, they are invoked by the microprocessor whenever there are conditions (exceptions) that the CPU is unable to handle.
- ✓ One such situation is an attempt to divide a number by zero. Since the result of dividing a number by zero is undefined, and the CPU has no way of handling such a result, it automatically invokes the divide error exception interrupt.
- ✓ In the 8088/86 microprocessor, out of 256 interrupts, Intel has set aside only INT 0 for the exception interrupt.
- ✓ INT 00 is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ✓ In the x86 PC, the service subroutine for this interrupt is responsible for displaying the message "DIVIDE ERROR" on the screen if a program such as the following is executed:

```
MOV   AL,92        ;AL=92
SUB   CL,CL        ;CL=0
DIV   CL           ;92/0=undefined result
```

- ✓ INT 0 is also invoked if the quotient is too large to fit into the assigned register when executing a DIV instruction. Look at the following case:

```
MOV    AX,0FFFFH      ;AX=FFFFH
MOV    BL,2           ;BL=2
DIV    BL             ;65535/2 = 32767 larger than 255
                      ;maximum capacity of AL
```

INT 01 (single step)

- ✓ In executing a sequence of instructions, there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. This is commonly referred to as single-stepping, or performing a trace.
- ✓ Intel has designated INT 01 specifically for implementation of single-stepping. To single-step, the trap flag (TF) (D8 of the flag register), must be set to 1. Then after execution of each instruction, the 8088/86 automatically jumps to physical location 00004 to fetch the 4 bytes for CS: IP of the interrupt service routine, which will dump the registers onto the screen.
- ✓ Intel has not provided any specific instruction for to set or reset (unlike IF, which uses STI and CLI instructions to set or reset), the TF; one can write a simple program to do that. The following shows how to make TF = 0:

```
PUSHF
POP    AX
AND    AX,1111111011111111B
PUSH   AX
POPF
```

- ✓ Recall that, TF is D8 of the flag register.
- ✓ To make TF = 1, one simply uses the OR instruction in place of the AND instruction above.

INT 02 (non-maskable interrupt)

- ✓ All Intel x86 microprocessors have a pin designated NMI. It is an active-high input. Intel has set aside INT 2 for the NMI interrupt. Whenever the NMI pin of the x86 is activated by a high (5 V) signal, the CPU jumps to physical memory location 00008 to fetch the CS: IP of the interrupt service routine associated with NMI.
- ✓ The NMI input is often used for major system faults, such as power failures. The NMI interrupt will be caused whenever AC power drops out. In response to this interrupt, the microprocessor stores all of the internal registers in a battery-backed-up memory or an EEPROM.

INT 03 (breakpoint)

- ✓ To allow implementation of breakpoints in software engineering, Intel has set aside INT 03.

- ✓ In single-step mode, one can inspect the CPU and system memory after the execution of each instruction, a breakpoint is used to examine the CPU and memory after the execution of a group of instructions.
- ✓ INT 3 is a 1-byte instruction; where as all other "*INT nn*" instructions are 2-byte instructions.

INT 04 (signed number overflow)

- ✓ This interrupt is invoked by a signed number overflow condition. There is an instruction associated with this, INTO (interrupt on overflow).
- ✓ The CPU will activate INT 04 if OF = 1. In cases, where OF = 0, the INTO instruction is not executed; but is bypassed and acts as a NOP (no operation) instruction.
- ✓ To understand this, look at the following example: Suppose in the following program; DATA1= +64 = 0100 0000 and DATA2 = +64 = 0100 0000. The INTO instruction will be executed and the 8088/86 will jump to physical location 00010H, the memory location associated with INT 04. The carry from D6 to D7 causes the overflow flag to become l.
- ✓ Now, the INTO causes the CPU to perform "*INT 4*" and jump to physical location 00010H of the vector table to get the CS: IP of the service routine.

```
MOV   AL,DATA1
MOV   BL,DATA2
ADD   AL,BL;add BL to AL
INTO
```

```
   + 64   0100 0000
+  + 64   0100 0000
   +128   1000 0000   OF=1 and the result is not +128
```

- ✓ Suppose that the data in the above program was DATA1 = +64 and DATA2 = +17. In that case, OF would become 0; the INTO is not executed and acts simply as a NOP (no operation) instruction.

<u>x86 PC AND INTERRUPT ASSIGNMENT:</u>

- Of the 256 possible interrupts in the x86;
  - ✓ some are used by the PC peripheral hardware (BIOS)
  - ✓ some are used by the Microsoft operating system
  - ✓ the rest are available for programmers of software applications.

For a given ISR, the logical address is F000:FF53. Verify that the physical address is FFF53H.

Solution:

Since the logical address is F000:FF53, this means that CS = F000H and IP = FF53H. Shifting left the segment register one hex digit and adding it to the offset gives the physical address FFF53H.

INT 21H & INT 10H PROGRAMMING

The INT instruction has the following format:

```
INT  xx;the interrupt number xx can be 00 - FFH
```

Interrupts are numbered 00 to FF; this gives a total of 256 interrupts in x86 microprocessors. Of these 256 interrupts, two of them are the most widely used: INT 10H and INT 21H.

BIOS INT 10H PROGRAMMING:

- INT 10H subroutines are burned into the ROM BIOS of the x86-based IBM PC and compatibles and are used to communicate with the computer's screen video. The manipulation of screen text or graphics can be done through INT 10H.
- There are many functions associated with INT 10H. Among them are changing the color of characters or the background color, clearing the screen, and changing the location of the cursor.
- These options are chosen by putting a specific value in register AH.

Monitor Screen in Text Mode:

- ✓ The monitor screen in the x86 PC is divided into 80 columns and 25 rows in normal text mode (see the following Fig). In other words, the text screen is 80 characters wide by 25 characters long.

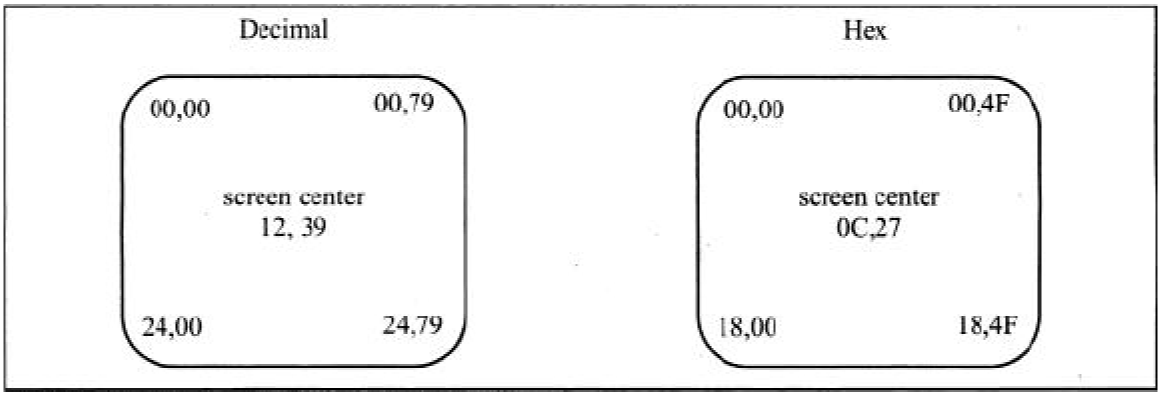


Fig: Cursor Locations (row, column)

- ✓ Since both a row and a column number are associated with each location on the screen, one can move the cursor to any location on the screen simply by changing the row and column values.
- ✓ The 80 columns are numbered from 0 to 79 and the 25 rows are numbered 0 to 24. The top left comer has been assigned 00, 00 (row = 00, column = 00). Therefore, the top right comer will be 00, 79 (row = 00, column = 79).
- ✓ Similarly, the bottom left comer is 24, 00 (row = 24, column = 00) and the bottom right corner of the monitor is 24, 79 (row = 24, column = 79).

INT 10H Function 06H: Clearing the Screen

To clear the screen before displaying data; the following registers must contain certain values before INT 10H is called: AH = 06, AL = 00, BH = 07, CX = 0000, DH = 24, and DL= 79. The code will look like this:

```
MOV   AH,06        ;AH=06 to select scroll function
MOV   AL,00        ;AL=00 the entire page
MOV   BH,07        ;BH=07 for normal attribute
MOV   CH,00        ;CH=00 row value of start point
MOV   CL,00        ;CL=00 column value of start point
MOV   DH,24        ;DH=24 row value of ending point
MOV   DL,79        ;DL=79 column value of ending point
INT   10H          ;invoke the interrupt
```

- ✓ Remember that DEBUG assumes immediate operands to be in hex; therefore, DX would be entered as 184F. However, MASM assumes immediate operands to be in decimal. In that case DH = 24 and DL = 79.
- ✓ In the program above, one of many options of INT 10H was chosen by putting 06 into AH. Option AH = 06, called the scroll function, will cause the screen to scroll upward.
- ✓ The CH and CL registers hold the starting row and column, respectively, and DH and DL hold the ending row and column.
- ✓ To clear the entire screen, one must use the top left cursor position of 00, 00 for the start point and the bottom right position of 24, 79 for the end point.
- ✓ Option AH = 06 of INT 10H is in reality the "*scroll window up*" function; therefore, one could use that to make a window of any size by choosing appropriate values for the start and end rows and columns.

- ✓ To clear the screen, the top left and bottom right values are used for start and stop points in order to scroll up the entire screen. It is more efficient coding to clear the screen by combining some of the lines above as follows:

```
MOV   AX,0600H     ;scroll entire screen
MOV   BH,07        ;normal attribute
MOV   CX,0000      ;start at 00,00
MOV   DX,184FH     ;end at 24,79 (hex = 18,4F)
INT   10H          ;invoke the interrupt
```

INT 10H Function 02: Setting the Cursor to a Specific Location

- ✓ INT 10H function AH = 02 will change the position of the cursor to any location.
- ✓ The desired position of the cursor is identified by the row and column values in DX, where DH = row and DL = column.
- ✓ Video RAM can have multiple pages of text, but only one of them can be viewed at a time. When AH = 02, to set the cursor position, page zero is chosen by making BH = 00.

Write the code to set the cursor position to row = 15 = 0FH and column = 25 = 19H.

Solution:

```
MOV   AH,02        ;set cursor option
MOV   BH,00        ;page 0
MOV   DL,25        ;column position
MOV   DH,15        ;row position
INT   10H          ;invoke interrupt 10H
```

Write a program that (1) clears the screen and (2) sets the cursor at the center of the screen.

Solution:

The center of the screen is the point at which the middle row and middle column meet. Row 12 is at the middle of rows 0 to 24 and column 39 (or 40) is at the middle of columns 0 to 79. By setting row = DH = 12 and column = DL = 39, the cursor is set to the screen center.

```
;clearing the screen
      MOV   AX,0600H     ;scroll the entire page
      MOV   BH,07        ;normal attribute
      MOV   CX,0000      ;row and column of top left
      MOV   DX,184FH     ;row and column of bottom right
      INT   10H          ;invoke the video BIOS service

;setting the cursor to the center of screen
      MOV   AH,02        ;set cursor option
      MOV   BH,00        ;page 0
      MOV   DL,39        ;center column position
      MOV   DH,12        ;center row position
      INT   10H          ;invoke interrupt 10H
```

INT 10H Function 03: Get Current Cursor Position

In text mode, it is possible to determine where the cursor is located at any time by executing the following:

```
MOV   AH,03        ;option 03 of BIOS INT 10H
MOV   BH,00        ;page 00
INT   10H          ;interrupt 10H routine
```

- ✓ After execution of the program above, registers DH and DL will have the current row and column positions, and CX provides information about the shape of the cursor.
- ✓ The reason that page 00 was chosen is that the video memory could contain more than one page of data, depending on the video board installed on the PC.
- ✓ In text mode, page 00 is chosen for the currently viewed page.

Attribute Byte in Monochrome Monitors:

- ✓ There is an attribute associated with each character on the screen.

- ✓ The attribute provides information to the video circuitry, such as color and intensity of the character (foreground) and the background.
- ✓ The attribute byte for each character on the monochrome monitor is limited. The following Fig shows bit definitions of the monochrome attribute byte.

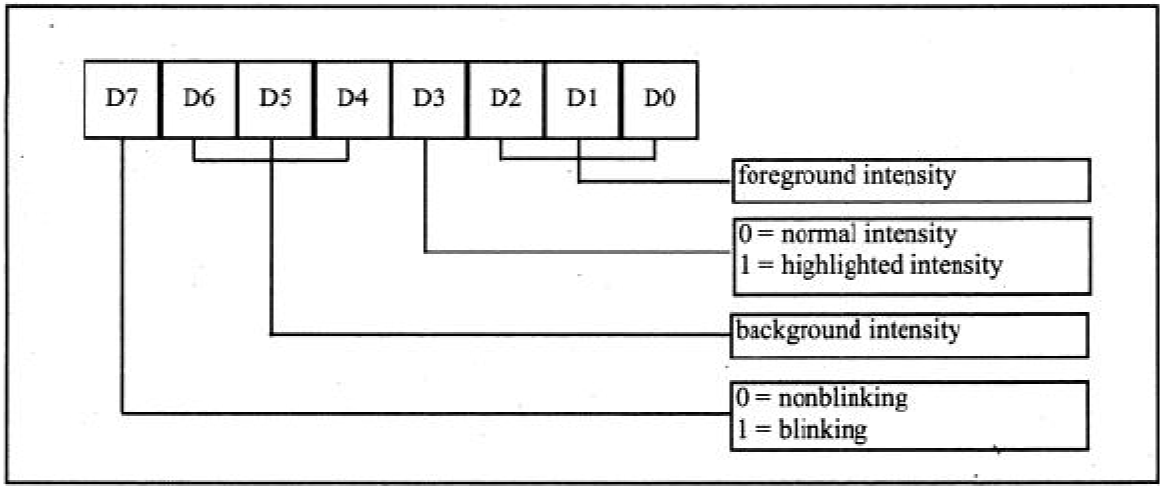


Fig: Attribute Byte for Monochrome Monitors

The following are some possible variations of the attributes shown in the above Fig.

Binary	Hex	Result
0000 0000	00	white on white (no display)
0000 0111	07	white on black normal
0000 1111	0F	white on black highlight
1000 0111	87	white on black blinking
0111 0111	77	black on black (no display)
0111 0000	70	black on white
1111 0000	F0	black on white blinking

Write a program using INT 10H to:
(a) Change the video mode.
(b) Display the letter "D" in 200H locations with attributes black on white blinking (blinking letters "D" are black and the screen background is white).
(c) Then use DEBUG to run and verify the program.

Solution:

(a) INT 10H function AH = 00 is used with AL = video mode to change the video mode. Use AL = 03.

```
MOV   AH,00        ;SET MODE OPTION
MOV   AL,03        ;CHANGE THE VIDEO MODE
INT   10H          ;MODE OF 80X25 FOR ANY COLOR MONITOR
```

(b) With INT 10H function AH = 09, one can display a character a certain number of times with specific attributes.

```
MOV   AH,09       ;DISPLAY OPTION
MOV   BH,00       ;PAGE 0
MOV   AL,44H      ;THE ASCII FOR LETTER "D"
MOV   CX,200H     ;REPEAT IT 200H TIMES
MOV   BL,0F0H     ;BLACK ON WHITE BLINKING
INT   10H
```

(c) Reminder: DEBUG assumes that all the numbers are in hex.

```
C>debug
-A
1131:0100 MOV AH,00
1131:0102 MOV AL,03    ;CHANGE THE VIDEO MODE
1131:0104 INT 10
1131:0106 MOV AH,09
1131:0108 MOV BH,00
1131:010A MOV AL,44
1131:010C MOV CX,200
1131:010F MOV BL,F0
1131:0111 INT 10
1131:0113 INT 3
1131:0114
-
```

Now see the result by typing in the command -G. Make sure that IP = 100 before running it. As an exercise, change the BL register to other attribute values given earlier. For example, BL = 07 white on black, or BL = 87H white on black blinking.

Attribute Byte in CGA Text Mode:

The bit definition of the attribute byte in CGA text mode is shown in the following Fig.

D7	D6	D5	D4	D3	D2	D1	D0
B	R	G	B	I	R	G	B
	background				foreground		

B = blinking
I = foreground intensity
Blinking and intensity apply to foreground only.

From the bit definition, it can be seen that, the background can take eight different colors by combining the prime colors red, blue, and green. The foreground can be any of 16 different colors by combining red, blue, green, and intensity.

Binary	Hex	Color effect
0000 0000	00	Black on black
0000 0001	01	Blue on black
0001 0010	12	Green on blue
0001 0100	14	Red on blue
0001 1111	1F	High-intensity white on blue

The following Program shows the use of the attribute byte in CGA mode.

Write a program that puts 20H (ASCII space) on the entire screen. Use high-intensity white on a blue background attribute for any characters to be displayed.

Solution:

```
MOV   AH,00        ;SET MODE OPTION
MOV   AL,03        ;CGA COLOR TEXT MODE OF 80 × 25
INT   10H
MOV   AH,09        ;DISPLAY OPTION
MOV   BH,00        ;PAGE 0
MOV   AL,20H       ;ASCII FOR SPACE
MOV   CX,800H      ;REPEAT IT 800H TIMES
MOV   BL,1FH       ;HIGH-INTENSITY WHITE ON BLUE
INT   10H
```

Graphics: Pixel Resolution and Color:

- In the text mode, the screen is viewed as a matrix of rows and columns of characters.
- In graphics mode, the screen is viewed as a matrix of horizontal and vertical pixels.
- The number of pixels varies among monitors and depends on monitor resolution and the video board.
- There are two facts associated with every pixel on the screen:
  - ✓ The location of the pixel
  - ✓ Its attributes, color, and intensity
- These two facts must be stored in the video RAM.
- Higher the number of pixels and colors, the larger the amount of memory is needed to store.
- The CGA mode can have a maximum of 16K bytes of video memory.
- This 16K bytes of memory can be used in three different ways:
  - ✓ Text mode of 80 x 25 characters: Use AL = 03 for mode selection in INT 10H option AH = 00. In this mode, 16 colors are supported.
  - ✓ Graphics mode of resolution 320 x 200 (medium resolution): Use AL = 04. In this mode, 4 colors are supported.
  - ✓ Graphics mode of resolution 640 x 200 (high resolution): Use AL = 06. In this mode, only 1 color (black and white) is supported.

- Hence, with a fixed amount of video RAM, the number of supported colors decreases as the resolution increases.

Table: The 16 Possible Colors

I	R	G	B	Color		I	R	G	B	Color
0	0	0	0	Black		1	0	0	0	Gray
0	0	0	1	Blue		1	0	0	1	Light Blue
0	0	1	0	Green		1	0	1	0	Light Green
0	0	1	1	Cyan		1	0	1	1	Light Cyan
0	1	0	0	Red		1	1	0	0	Light Red
0	1	0	1	Magenta		1	1	0	1	Light Magenta
0	1	1	0	Brown		1	1	1	0	Yellow
0	1	1	1	White		1	1	1	1	High Intensity White

INT 10H and Pixel Programming:

To draw a horizontal line, choose values for the row and column to point to the beginning of the line and then continue to increment the column until it reaches the end of the line, as shown in Example below:

Write a program to: (a) clear the screen, (b) set the mode to CGA of 640 × 200 resolution, and (c) draw a horizontal line starting at column = 100, row = 50, and ending at column 200, row 50.

Solution:

```
        MOV   AX,0600H    ;SCROLL THE SCREEN
        MOV   BH,07       ;NORMAL ATTRIBUTE
        MOV   CX,0000     ;FROM ROW=00,COLUMN=00
        MOV   DX,184FH    ;TO ROW=18H,COLUMN=4FH
        INT   10H         ;INVOKE INTERRUPT TO CLEAR SCREEN
        MOV   AH,00       ;SET MODE
        MOV   AL,06       ;MODE = 06 (CGA HIGH RESOLUTION)
        INT   10H         ;INVOKE INTERRUPT TO CHANGE MODE
        MOV   CX,100      ;START LINE AT COLUMN =100 AND
        MOV   DX,50       ;ROW = 50
BACK:   MOV   AH,0CH      ;AH=0CH TO DRAW A LINE
        MOV   AL,01       ;PIXELS = WHITE
        INT   10H         ;INVOKE INTERRUPT TO DRAW LINE
        INC   CX          ;INCREMENT HORIZONTAL POSITION
        CMP   CX,200      ;DRAW LINE UNTIL COLUMN = 200
        JNZ   BACK
```

DOS INTERRUPT 21H:

- INT21H is provided by DOS, which is BIOS-ROM based.
- When the OS is loaded into the computer, INT 21H can be invoked to perform some extremely useful functions. These functions are commonly referred to as *DOS INT 21H* function calls.

INT 21H Option 09: Outputting a String of Data to the Monitor

- ✓ INT 21H can be used to send a set of ASCII data to the monitor. To do that, the following registers must be set: AH = 09 and DX = the offset address of the ASCII data to be displayed.
- ✓ The address in the DX register is an offset address and DS is assumed to be the data segment. INT 21H option 09 will display the ASCII data string pointed at by DX until it encounters the dollar sign "$".
- ✓ In the absence of encountering a dollar sign, DOS function call 09 will continue to display any garbage that it can find in subsequent memory locations until it finds "$".

```
DATA_ASC    DB    'The earth is but one country','$'

      MOV   AH,09          ;option 09 to display string of data
      MOV   DX,OFFSET DATA_ASC      ;DX= offset address of data
      INT   21H                     ;invoke the interrupt
```

INT 21H Option 02: Outputting a Single Character to the Monitor

- ✓ To output a single character to the monitor, 02 is put in AH, DL is loaded with the character to be displayed, and then INT 21H is invoked. The following displays the letter "J'.

```
MOV   AH,02       ;option 02 displays one character
MOV   DL,'J'      ;DL holds the character to be displayed
INT   21H         ;invoke the interrupt
```

INT 21H Option 01: Inputting a Single Character, with Echo

This function waits until a character is input from the keyboard, and then echoes it to the monitor. After the interrupt, the input character (ASCII value) will be in AL.

```
MOV   AH,01 ;option 01 inputs one character
INT   21H   ;after the interrupt, AL = input character (ASCII)
```

The Program 4-1 does the following:

1. clears the screen
2. sets the cursor to the center of the screen, and
3. starting at that point of the screen, displays the message "This is a test of the display routine".

```
TITLE       PROG4-1 SIMPLE DISPLAY PROGRAM
PAGE        60,132
            .MODEL SMALL
            .STACK      64
;-----------------------------
            .DATA
MESSAGE     DB    'This is a test of the display routine','$'
;-----------------------------
     .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     CALL   CLEAR              ;CLEAR THE SCREEN
     CALL   CURSOR             ;SET CURSOR POSITION
     CALL   DISPLAY            ;DISPLAY MESSAGE
     MOV    AH,4CH
     INT    21H                ;GO BACK TO DOS
MAIN ENDP
;-----------------------------
;
;THIS SUBROUTINE CLEARS THE SCREEN
CLEAR PROC
     MOV    AX,0600H           ;SCROLL SCREEN FUNCTION
     MOV    BH,07              ;NORMAL ATTRIBUTE
     MOV    CX,0000            ;SCROLL FROM ROW=00,COL=00
     MOV    DX,184FH           ;TO ROW=18H,COL=4FH
     INT    10H                ;INVOKE INTERRUPT TO CLEAR SCREEN
     RET
CLEAR ENDP
;-----------------------------
;THIS SUBROUTINE SETS THE CURSOR AT THE CENTER OF THE SCREEN
CURSOR PROC
     MOV    AH,02              ;SET CURSOR FUNCTION
     MOV    BH,00              ;PAGE 00
     MOV    DH,12              ;CENTER ROW
     MOV    DL,39              ;CENTER COLUMN
     INT    10H                ;INVOKE INTERRUPT TO SET CURSOR POSITION
     RET
CURSOR ENDP
;-----------------------------
;THIS SUBROUTINE DISPLAYS A STRING ON THE SCREEN
DISPLAY  PROC
     MOV    AH,09              ;DISPLAY FUNCTION
     MOV    DX,OFFSET MESSAGE  ;DX POINTS TO OUTPUT BUFFER
     INT    21H                ;INVOKE INTERRUPT TO DISPLAY STRING
     RET
DISPLAY  ENDP
     END    MAIN
```

Program 4-1

INT 21H Option 0AH: Inputting a String of Data from the Keyboard

- ✓ Option 0AH of INT 21H provides a means by which one can get data from the keyboard and store it in a predefined area of memory in the data segment.
- ✓ To do this; the register options are: AH = 0AH and DX = offset address at which the string of data is stored.
- ✓ This is commonly referred to as a buffer area.

- ✓ DOS requires that a buffer area be defined in the data segment and the first byte specifies the size of the buffer. DOS will put the number of characters that came in through the keyboard in the second byte and the keyed-in data is placed in the buffer starting at the third byte.
- ✓ For example, the following program will accept up to six characters from the keyboard, including the return (carriage return) key. Six locations were reserved for the buffer and filled with FFH.
- ✓ The following shows portions of the data segment and code segment:

```
ORG    0010H
DATA1 DB    6,?,6 DUP (FF);0010H=06, 0012H to 0017H = FF

       MOV   AH,0AH            ;string input option of INT 21H
       MOV   DX,OFFSET DATA1   ;load the offset address of buffer
       INT   21H               ;invoke interrupt 21H
```

- ✓ The following shows the memory contents of offset 0010H:

```
0010  0011  0012  0013  0014  0015  0016  0017
06    00    FF    FF    FF    FF    FF    FF
```

- ✓ When this program is executed, the computer waits for the information to come in from the keyboard.
- ✓ When the data comes in, the IBM PC will not exit the INT 21H routine until it encounters the return key.
- ✓ Assuming the data that was entered through the keyboard was "USA" <RETURN>, the contents of memory locations starting at offset 0010H would look like this:

```
0010  0011  0012  0013  0014  0015  0016  0017
06    03    55    53    41    0D    FF    FF
USACR
```

- ✓ The step-by-step analysis is given below:

0010H = 06	DOS requires the size of the buffer in the first location.
0011H = 03	The keyboard was activated three times (excluding the RETURN key) to key in the letters U, S, and A.
0012H = 55H	This is the ASCII hex value for letter U.
0013H = 53H	This is the ASCII hex value for letter S.
0014H = 41H	This is the ASCII hex value for letter A.
0015H = 0DH	This is the ASCII hex value for CR (carriage return).

- ✓ The 0AH option of INT 21H accepts the string of data from the keyboard and echoes (displays) it on the screen as it is keyed in.

Use of Carriage Return and Line Feed:

- o In the Program 4-2, the EQU statement is used to equate CR (carriage return) with its ASCII value of 0DH, and LF (line feed) with its ASCII value of 0AH.
- o This makes the program much more readable. Since the result of the conversion was to be displayed in the next line, the string was preceded by CR and LF.

- In the absence of CR the string would be displayed wherever the cursor happened to be.
- In the case of CR and no LF, the string would be displayed on the same line after it had been returned to the beginning of the line.

```
;Program 4-2 performs the following: (1) clears the screen, (2) sets
;the cursor at the beginning of the third line from the top of the
;screen, (3) accepts the message "IBM perSonal COmputer" from the
;keyboard, (4) converts lowercase letters of the message to uppercase,
;(5) displays the converted results on the next line.

 TITLE      PROG4-2
 PAGE       60,132
            .MODEL SMALL
            .STACK      64

 ;----------------------------
            .DATA
 BUFFER     DB     22,?,22 DUP (?)        ;BUFFER FOR KEYED-IN DATA
            ORG    18H
 DATAREA    DB     CR,LF,22 DUP (?),'$'   ;DATA HERE AFTER CONVERSION
 ;DTSEG     ENDS
 CR    EQU 0DH
 LF    EQU 0AH

 ;----------------------------
       .CODE
 MAIN PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  CLEAR               ;CLEAR THE SCREEN
      CALL  CURSOR              ;SET CURSOR POSITION
      CALL  GETDATA             ;INPUT A STRING INTO BUFFER
      CALL  CONVERT             ;CONVERT STRING TO UPPERCASE
      CALL  DISPLAY             ;DISPLAY STRING DATAREA
      MOV   AH,4CH
      INT   21H         ;GO BACK TO DOS
 MAIN ENDP

 ;-----------------------------
 ;THIS SUBROUTINE CLEARS THE SCREEN
 CLEAR PROC
      MOV   AX,0600H            ;SCROLL SCREEN FUNCTION
      MOV   BH,07               ;NORMAL ATTRIBUTE
      MOV   CX,0000                   ;SCROLL FROM ROW=00,COL=00
      MOV   DX,184FH            ;TO ROW=18H,4FH
      INT   10H                 ;INVOKE INTERRUPT TO CLEAR SCREEN
      RET
 CLEARENDP
 ;THIS SUBROUTINE SETS THE CURSOR TO THE BEGINNING OF THE 3RD LINE
 CURSOR PROC
      MOV   AH,02               ;SET CURSOR FUNCTION
      MOV   BH,00               ;PAGE 0
      MOV   DL,01               ;COLUMN 1
      MOV   DH,03               ;ROW 3
      INT   10H                 ;INVOKE INTERRUPT TO SET CURSOR
      RET
 CURSOR ENDP

 ;-----------------------------
```

```
;THIS SUBROUTINE DISPLAYS A STRING ON THE SCREEN
DISPLAY PROC
     MOV   AH,09              ;DISPLAY STRING FUNCTION
     MOV   DX,OFFSET DATAREA ;DX POINTS TO BUFFER
     INT   21H                ;INVOKE INTERRUPT TO DISPLAY STRING
     RET
DISPLAY    ENDP

;------------------------------
;THIS SUBROUTINE PUTS DATA FROM THE KEYBOARD INTO A BUFFER
GETDATA PROC
     MOV   AH,0AH             ;INPUT STRING FUNCTION
     MOV   DX,OFFSET BUFFER   ;DX POINTS TO BUFFER
     INT   21H                ;INVOKE INTERRUPT TO INPUT STRING
     RET
GETDATA ENDP
;------------------------------
;THIS SUBROUTINE CONVERTS ANY SMALL LETTER TO ITS CAPITAL
CONVERT PROC
     MOV   BX,OFFSET BUFFER
     MOV   CL,[ BX] +1                ;GET THE CHAR COUNT
     SUB   CH,CH                      ;CX = TOTAL CHARACTER COUNT
     MOV   DI,CX                      ;INDEXING INTO BUFFER
     MOV   BYTE PTR[ BX+DI] +2,20H    ;REPLACE CR WITH SPACE
     MOV   SI,OFFSET DATAREA+2        ;STRING ADDRESS
AGAIN: MOV AL,[ BX] +2                ;GET THE KEYED-IN DATA
     CMP   AL,61H                     ;CHECK FOR 'a'
     JB    NEXT                       ;IF BELOW, GO TO NEXT
     CMP   AL,7AH                     ;CHECK FOR 'z'
     JA    NEXT                       ;IF ABOVE GO TO NEXT
     AND   AL,11011111B               ;CONVERT TO CAPITAL
NEXT: MOV  [ SI] ,AL                  ;PLACE IN DATA AREA
     INC   SI                         ;INCREMENT POINTERS
     INC   BX
     LOOP  AGAIN                      ;LOOP IF COUNTER NOT ZERO
     RET
CONVERT ENDP
     END   MAIN
```

Program 4-2

- The Program 4-3 prompts the user to type in a name. The name can have a maximum of eight letters.
- After the name is typed in, the program gets the length of the name and prints it to the screen.

```
TITLE      PROG4-3     READS IN LAST NAME AND DISPLAYS LENGTH
PAGE       60,132
           .MODEL SMALL
           .STACK 64 (?)
;----------------------------
```

```
             .DATA
MESSAGE1     DB    'What is your last name?','$'
             ORG   20H
BUFFER1      DB    9,?,9 DUP (0)
             ORG   30H
MESSAGE2     DB    CR,LF,'The number of letters in your name is: ','$'
ROW          EQU 08
COLUMN       EQU 05
CR           EQU 0DH      ;EQUATE CR WITH ASCII CODE FOR CARRIAGE RETURN
LF           EQU 0AH      ;EQUATE LF WITH ASCII CODE FOR LINE FEED
;----------------------------
             .CODE
MAIN         PROC  FAR
     MOV     AX,@DATA
     MOV     DS,AX
     CALL    CLEAR
     CALL    CURSOR
     MOV     AH,09                 ;DISPLAY THE PROMPT
     MOV     DX,OFFSET MESSAGE1
     INT     21H
     MOV     AH,0AH                ;GET LAST NAME FROM KEYBOARD
     MOV     DX,OFFSET BUFFER1
     INT     21H
     MOV     BX,OFFSET BUFFER1 ;FIND OUT NUMBER OF LETTERS IN NAME
     MOV     CL,[ BX+1]            ;GET NUMBER OF LETTERS
     OR      CL,30H                ;MAKE IT ASCII
     MOV     MESSAGE2+40,CL        ;PLACE AT END OF STRING
     MOV     AH,09                 ;DISPLAY SECOND MESSAGE
     MOV     DX,OFFSET MESSAGE2
     INT     21H
     MOV     AH,4CH
     INT     21H          ;GO BACK TO DOS
MAIN ENDP
;-----------------------------
CLEAR PROC                         ;CLEAR THE SCREEN
     MOV     AX,0600H
     MOV     BH,07
     MOV     CX,0000
     MOV     DX,184FH
     INT     10H
     RET
CLEAR ENDP
;-----------------------------
CURSOR PROC                              ;SET CURSOR POSITION
     MOV     AH,02
     MOV     BH,00
     MOV     DL,COLUMN
     MOV     DH,ROW
     INT     10H
     RET
CURSOR ENDP
     END     MAIN
```

Program 4-3

- Program 4-4 demonstrates many of the functions described:

Write a program to perform the following: (1) clear the screen, (2) set the cursor at row 5 and column 1 of the screen, (3) prompt "There is a message for you from Mr. Jones. To read it enter Y ". If the user enters 'Y' or 'y' then the message "Hi! I must leave town tomorrow, therefore I will not be able to see you" will appear on the screen. If the user enters any other key, then the prompt "No more messages for you" should appear on the next line.

```
TITLE  PROGRAM 4-4
PAGE  60,132
      .MODEL SMALL
      .STACK 64
;----------------------------
             .DATA
PROMPT1      DB    'There is a message for you from Mr. Jones. '
             DB    'To read it enter Y','$'
MESSAGE      DB    CR,LF,'Hi! I must leave town tomorrow, '
             DB    'therefore I will not be able to see you','$'
PROMPT2      DB    CR,LF,'No more messages for you','$'
;DTSEG       ENDS
CR           EQU  0DH
LF           EQU  0AH
;----------------------------
      .CODE
MAIN  PROC   FAR
      MOV    AX,@DATA
      MOV    DS,AX
      CALL   CLEAR         ;CLEAR THE SCREEN
      CALL   CURSOR        ;SET CURSOR POSITION
      MOV    AH,09         ;DISPLAY THE PROMPT
      MOV    DX,OFFSET PROMPT1
      INT    21H
      MOV    AH,07         ;GET ONE CHAR, NO ECHO
      INT    21H
      CMP    AL,'Y'        ;IF 'Y', CONTINUE
      JZ     OVER
      CMP    AL,'y'
      JZ     OVER
      MOV    AH,09         ;DISPLAY SECOND PROMPT IF NOT Y
      MOV    DX,OFFSET PROMPT2
      INT    21H
      JMP    EXIT
OVER:MOV     AH,09         ;DISPLAY THE MESSAGE
      MOV    DX,OFFSET MESSAGE
      INT    21H
EXIT:MOV     AH,4CH
      INT    21H           ;GO BACK TO DOS
MAIN         ENDP
;------------------------------
CLEAR  PROC                ;CLEARS THE SCREEN
      MOV    AX,0600H
      MOV    BH,07
      MOV    CX,0000
      MOV    DX,184FH
      INT    10H
      RET
CLEAR        ENDP
;------------------------------
```

```
CURSOR PROC                    ;SET CURSOR POSITION
       MOV    AH,02
       MOV    BH,00
       MOV    DL,05        ;COLUMN 5
       MOV    DH,08        ;ROW 8
       INT    10H
       RET
CURSOR ENDP
       END    MAIN
```

Program 4-4

INT 21H Option 07: Keyboard Input without Echo

- ✓ Option 07 of INT 21H requires the user to enter a single character but that character is not displayed (or echoed) on the screen.
- ✓ After execution of the interrupt, the PC waits until a single character is entered and provides the character in AL.

```
MOV    AH,07 ;keyboard input without echo
INT    21H
```

Using the LABEL Directive to Define a String Buffer:

- o A more systematic way of defining the buffer area for the string input is to use the LABEL directive.
- o The LABEL directive can be used in the data segment to assign multiple names to data. When used in the data segment it looks like this:

```
name  LABEL attribute
```

- o The attribute can be BYTE, WORD, DWORD, FWORD, QWORD, or TBYTE.

```
JOE    LABEL BYTE
TOM    DB    20 DUP(0)
```

By: MAHESH PRASANNA K.,

DEPT. OF CSE, VCET.

****

****