MODULE – 3

SIGNED NUMBERS AND STRINGS & MEMORY INTERFACING & 8255

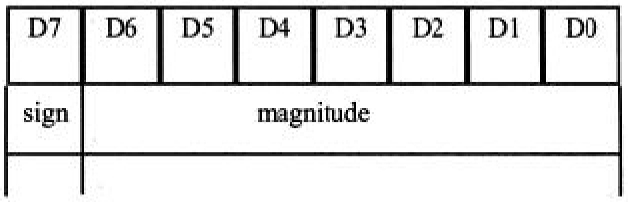
SIGNED NUMBERS & STRINGS

SIGNED NUMBER ARITHMETIC OPERATIONS:

- In everyday life, numbers are used that could be *positive or negative*. For example, a temperature of 5 degrees below zero can be represented as –5, and 20 degrees above zero as +20.
- Computers must be able to accommodate such numbers. To do that, an arrangement for the representation of signed positive and negative numbers is made:
  - ✓ The most significant bit (MSB) is set aside for the sign (+ or –)
  - ✓ The rest of the bits are used for the magnitude.
- The sign is represented by 0 for positive (+) numbers and 1 for negative (–) numbers.
- Note that, entire 8-bit or 16-bit operand will be treated as magnitude in the case of unsigned number representation.

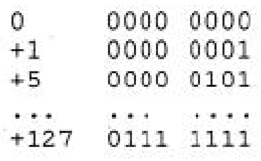
Byte-sized Signed Numbers:

- In signed byte operands, D7 (MSB) is the sign and D6 to D0 are set aside for the magnitude of the number.
  - ✓ If D7 = 0, the operand is positive
  - ✓ If 07 = 1, the operand is negative.



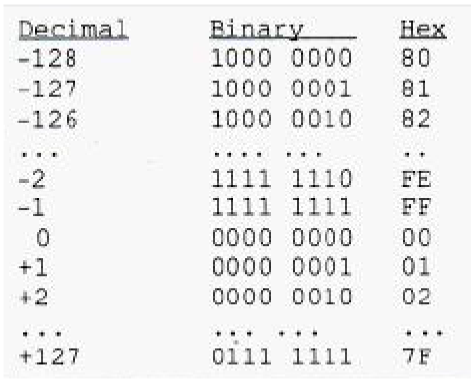
D7	D6	D5	D4	D3	D2	D1	D0
sign	magnitude						

- The range of *positive numbers* that can be represented by the format above is 0 to + 127.

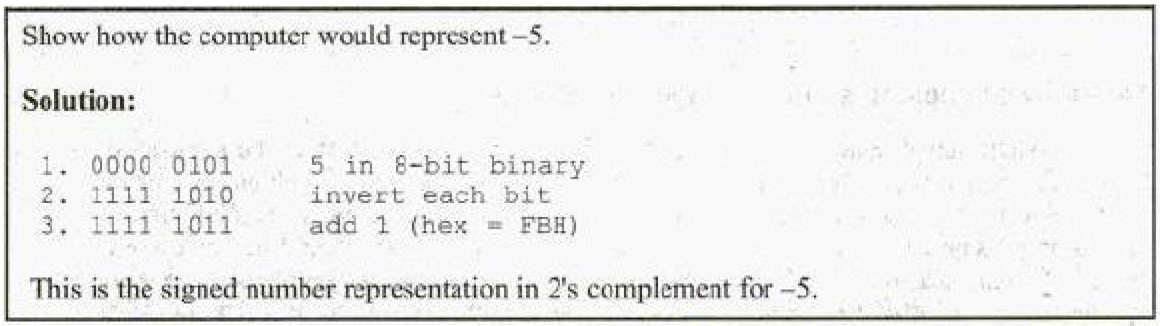


```
0       0000  0000
+1      0000  0001
+5      0000  0101
...     ...   ....
+127    0111  1111
```

- If a *positive number* is larger than +127, a word sized operand must be used.
- For *negative numbers* D7 is 1, but the magnitude is represented in 2's complement.
- Although the assembler does the conversion, it is still important to understand how the conversion works. To convert to negative number representation (2's complement), follow these steps:
  - ✓ Write the magnitude of the number in 8-bit binary (no sign).
  - ✓ Invert each bit
  - ✓ Add 1 to it.



Decimal	Binary	Hex
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
...	....	..
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
...	... ...	...
+127	0111 1111	7F

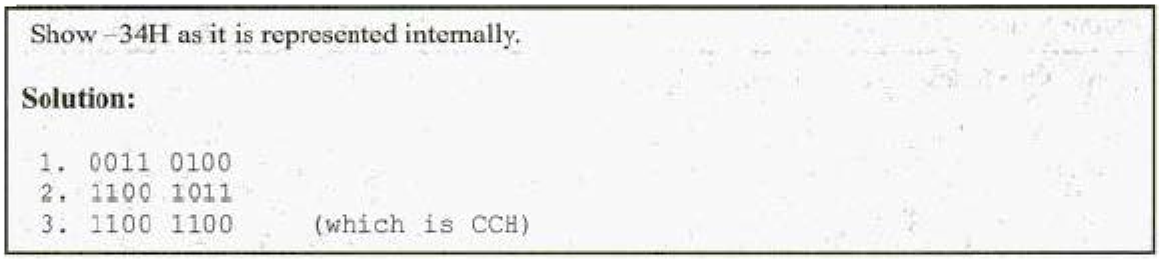


Show how the computer would represent –5.

Solution:

```
1. 0000 0101     5 in 8-bit binary
2. 1111 1010     invert each bit
3. 1111 1011     add 1 (hex = FBH)
```

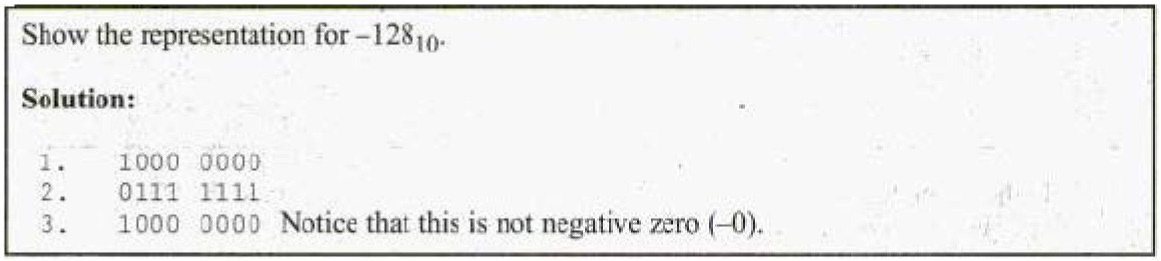
This is the signed number representation in 2's complement for –5.



Show –34H as it is represented internally.

Solution:

```
1. 0011 0100
2. 1100 1011
3. 1100 1100      (which is CCH)
```



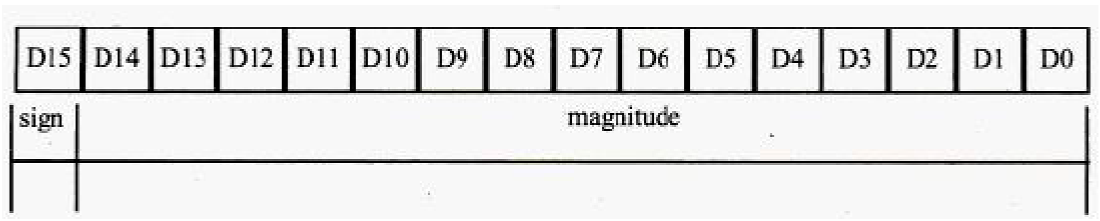
Show the representation for -128_{10}.

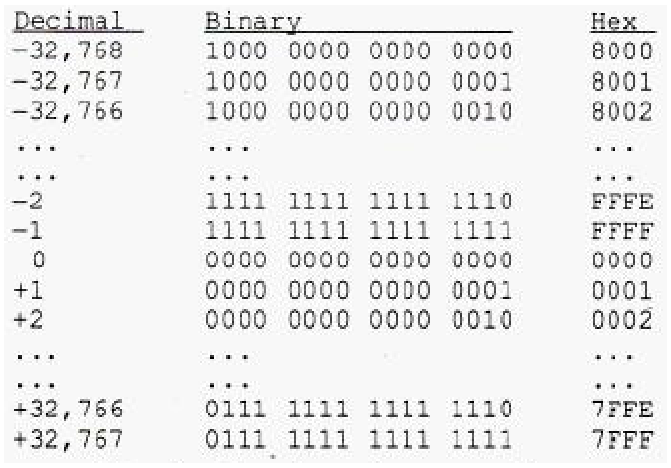
Solution:

```
1.   1000 0000
2.   0111 1111
3.   1000 0000  Notice that this is not negative zero (–0).
```

Word-sized Signed Numbers:

- In x86 computers a word is 16-bits in length. Setting aside the MSB (D15) for the sign leaves a total of 15 bits (D14 – D0) for the magnitude. This gives a range of –32,768 to +32,767.
- If a number is larger than this, it must be treated as a multiword operand and be processed chunk by chunk the same way as unsigned numbers.

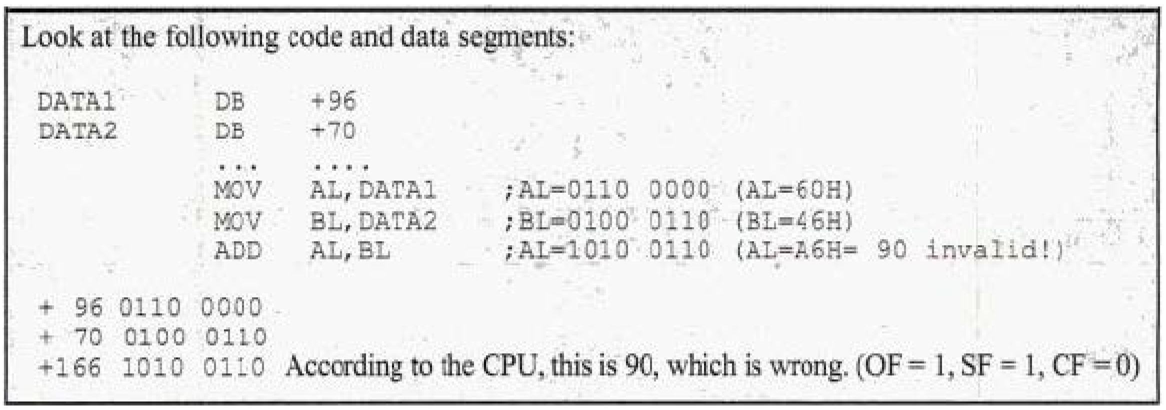




Decimal	Binary	Hex
-32,768	1000 0000 0000 0000	8000
-32,767	1000 0000 0000 0001	8001
-32,766	1000 0000 0000 0010	8002
...	...	...
...	...	...
-2	1111 1111 1111 1110	FFFE
-1	1111 1111 1111 1111	FFFF
0	0000 0000 0000 0000	0000
+1	0000 0000 0000 0001	0001
+2	0000 0000 0000 0010	0002
...	...	...
...	...	...
+32,766	0111 1111 1111 1110	7FFE
+32,767	0111 1111 1111 1111	7FFF

Overflow Problem in Signed Number Operations:

What is an overflow? If the result of an operation on signed numbers is too large for the register, an overflow occurs and the programmer must be notified. Look at following Example:



Look at the following code and data segments:

```
DATA1       DB     +96
DATA2       DB     +70
            ...    ....
            MOV    AL,DATA1     ;AL=0110 0000 (AL=60H)
            MOV    BL,DATA2     ;BL=0100 0110 (BL=46H)
            ADD    AL,BL        ;AL=1010 0110 (AL=A6H= 90 invalid!)

+ 96  0110 0000
+ 70  0100 0110
+166  1010 0110
```

According to the CPU, this is 90, which is wrong. (OF = 1, SF = 1, CF = 0)

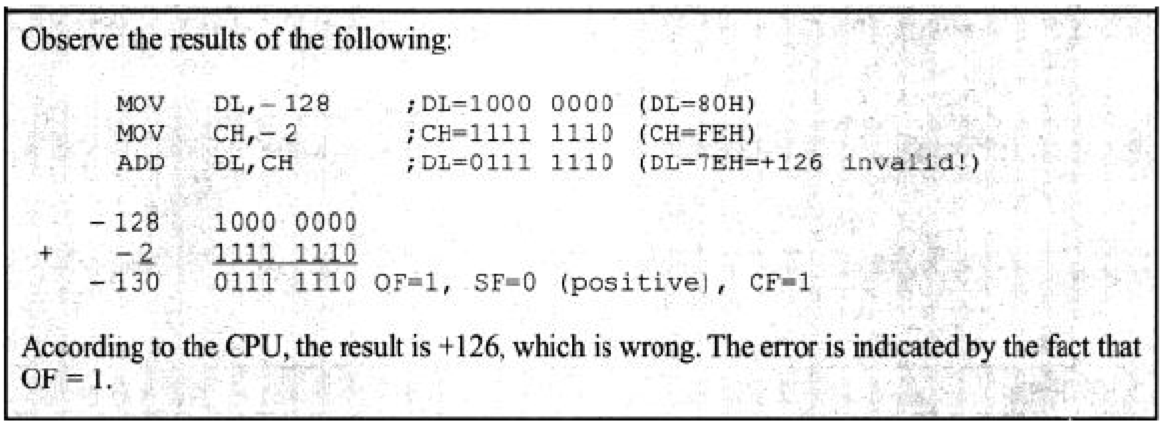
- In the example above; +96 is added to +70 and the result according to the CPU is –90 (5AH). Why?
- The reason is that, the result was more than what AL could handle. Like all other 8-bit registers, AL could only contain up to +127. The *designers of the CPU created the overflow flag specifically for the purpose of informing the programmer that the result of the signed number operation is erroneous.*

Hence, when using signed numbers, a serious problem with regarding overflow arises that must be dealt with. The CPU indicates the existence of the problem by raising the OF (overflow) flag, but it is up to the programmer to take care of it. The CPU understands only 0s and 1s and ignores the human convention of positive and negative numbers.

When Overflow Flag is Set in 8-bit Operations?

In 8-bit signed number operations, OF is set to 1, if either of the following two conditions occurs:

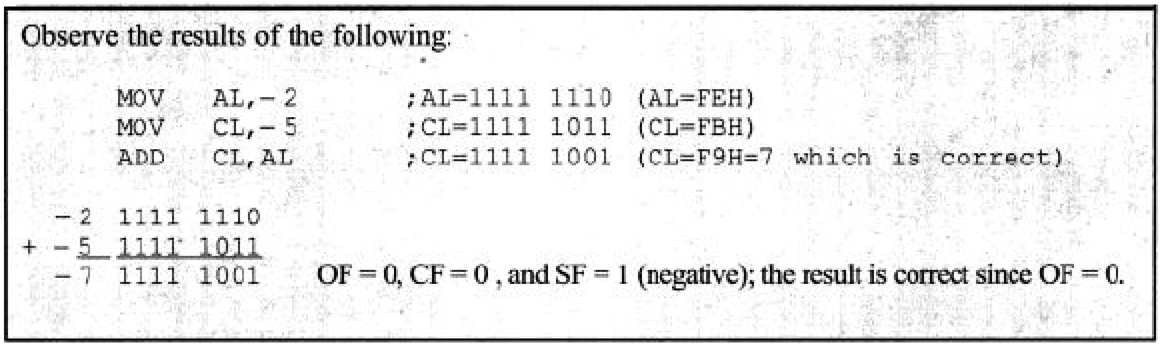
1. There is a carry from D6 to D7, but no carry out of D7 (CF = 0)
2. There is a carry from D7 out (CF = 1), but no carry from D6 to D7.



Observe the results of the following:

```
      MOV    DL,-128      ;DL=1000 0000  (DL=80H)
      MOV    CH,-2        ;CH=1111 1110  (CH=FEH)
      ADD    DL,CH        ;DL=0111 1110  (DL=7EH=+126 invalid!)

   -128    1000 0000
+    -2    1111 1110
   -130    0111 1110 OF=1, SF=0 (positive), CF=1
```

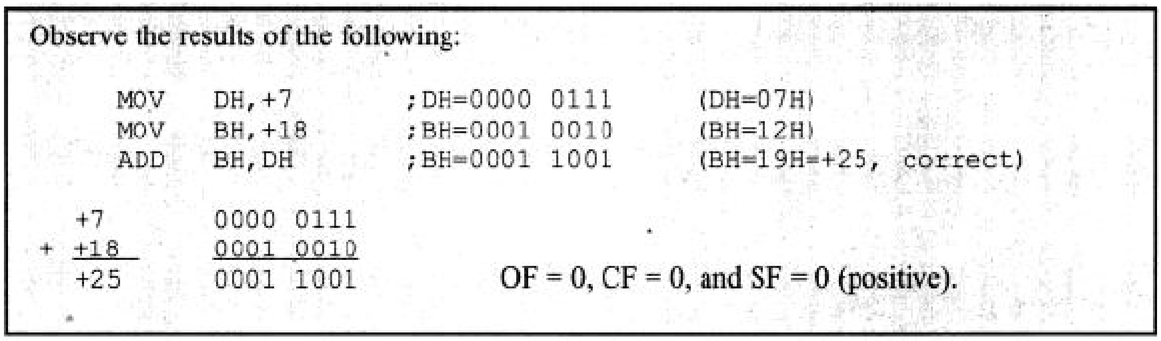
According to the CPU, the result is +126, which is wrong. The error is indicated by the fact that OF = 1.



Observe the results of the following:

```
      MOV    AL,-2        ;AL=1111 1110  (AL=FEH)
      MOV    CL,-5        ;CL=1111 1011  (CL=FBH)
      ADD    CL,AL        ;CL=1111 1001  (CL=F9H=7 which is correct)

   -2  1111 1110
+  -5  1111 1011
   -7  1111 1001
```

OF = 0, CF = 0 , and SF = 1 (negative); the result is correct since OF = 0.



Observe the results of the following:

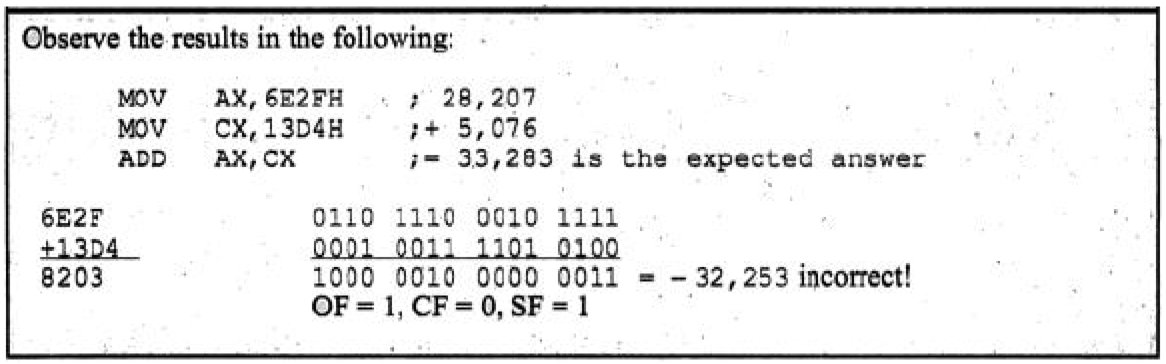
```
      MOV    DH,+7        ;DH=0000 0111      (DH=07H)
      MOV    BH,+18       ;BH=0001 0010      (BH=12H)
      ADD    BH,DH        ;BH=0001 1001      (BH=19H=+25, correct)

    +7       0000 0111
+  +18       0001 0010
   +25       0001 1001
```

OF = 0, CF = 0, and SF = 0 (positive).

When Overflow Flag is Set in 16-bit Operations?

In 16-bit signed number operations, OF is set to 1, if either of the following two conditions occurs:

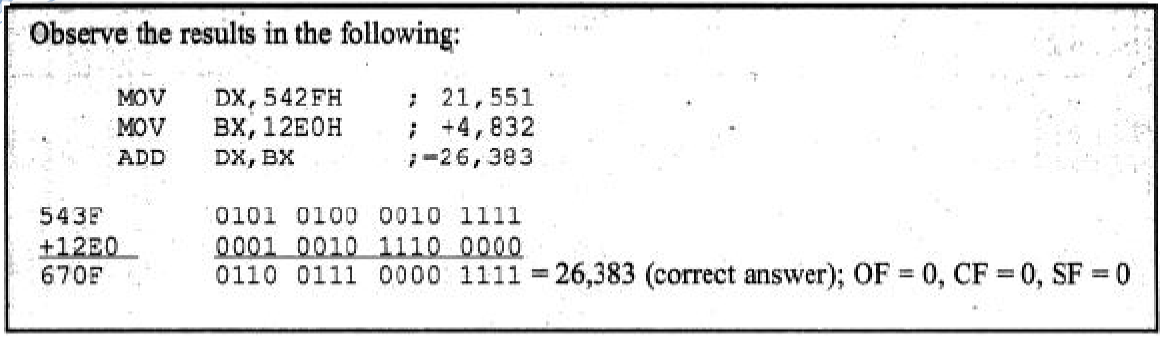
1. There is a carry from D14 to D15, but no carry out of D15 (CF = 0)
2. There is a carry from D15 out (CF = 1), but no carry from D14 to D15.



Observe the results in the following:

```
      MOV    AX,6E2FH     ; 28,207
      MOV    CX,13D4H     ;+ 5,076
      ADD    AX,CX        ;= 33,283 is the expected answer

  6E2F              0110 1110 0010 1111
 +13D4              0001 0011 1101 0100
  8203              1000 0010 0000 0011 = -32,253 incorrect!
```

OF = 1, CF = 0, SF = 1



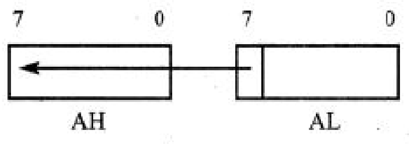
Observe the results in the following:

```
        MOV     DX,542FH     ; 21,551
        MOV     BX,12E0H     ; +4,832
        ADD     DX,BX        ;-26,383

543F            0101 0100 0010 1111
+12E0           0001 0010 1110 0000
670F            0110 0111 0000 1111
```

= 26,383 (correct answer); OF = 0, CF = 0, SF = 0

Avoiding Erroneous Results in Signed Number Operations:

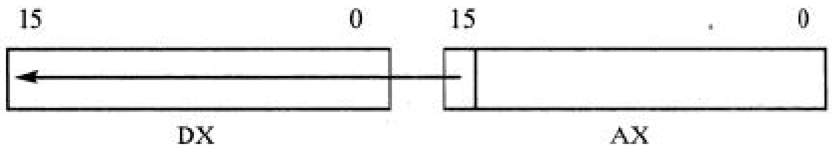
- To avoid the problems associated with signed number operations, one can sign extend the operand.
- Sign extension copies;
  - ✓ the sign bit (D7) of the lower byte of a register into the upper bits of the register, or
  - ✓ the sign bit of a 16-bit register into another register.
- The instructions used to perform the sign extension are;
- CBW (*convert signed byte to signed word*) – will copy D7 (the sign flag) of AL to all bit positions of AH register.



or:

```
MOV    AL,+96       ;AL=0110 0000
CBW                 ;now AH=0000 0000 and AL=0110 0000

MOV    AL,-2        ;AL=1111 1110
CBW                 ;AH=1111 1111 and AL=1111 1110
```

- CWD (*convert signed word to signed double word*): will copy D15 of AX to all bot positions of DX register.



example:

```
MOV    AX,+260      ;AX=0000 0001 0000 0100 or AX=0104H
CWD                 ;DX=0000H and AX=0104H
```

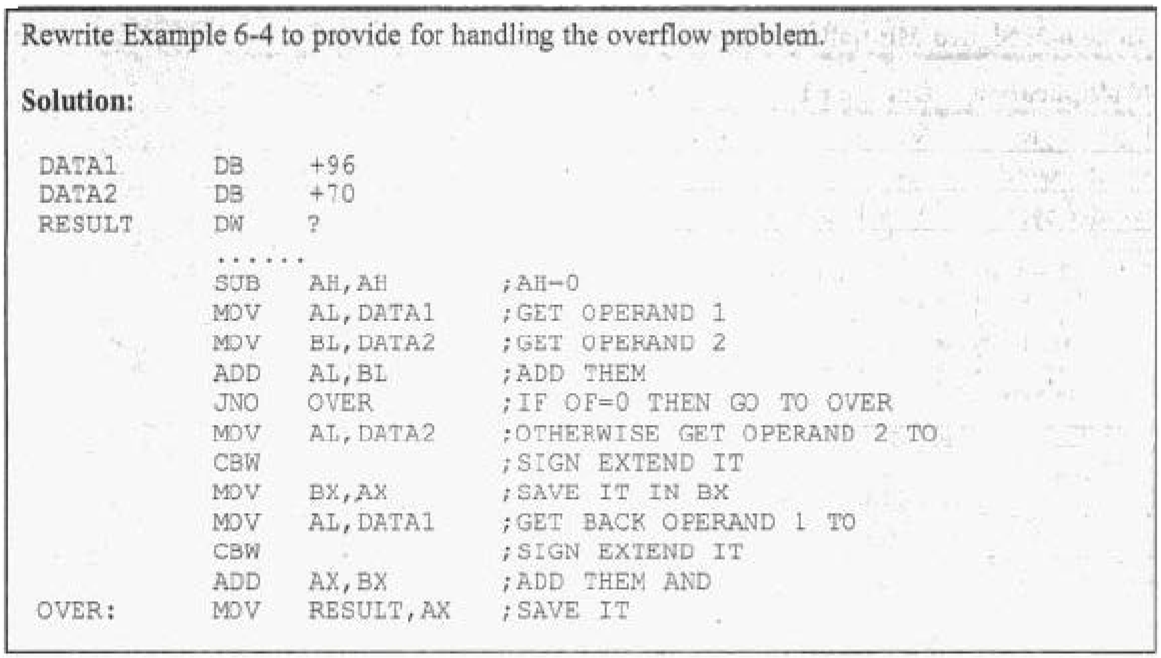
example:

```
MOV    AX,-32766    ;AX=1000 0000 0000 0010B or AX=8002H
CWD                 ;DX=FFFF and AX=8002
```

In the following Example (program for addition of any two signed bytes);

- ✓ If the overflow flag is not raised (OF = 0), the result of the signed number is correct and JNO (jump if no overflow) will jump to OVER.

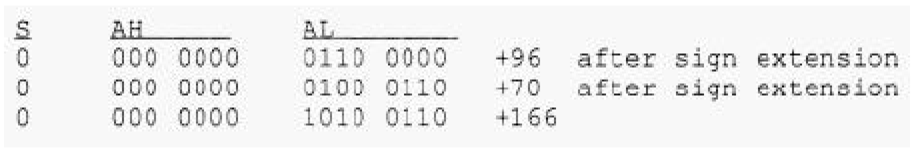
- ✓ If OF = 1, (which means that the result is erroneous), each operand must be sign extended and then added. That is the function of the code below the JNO instruction.



Rewrite Example 6-4 to provide for handling the overflow problem.

Solution:

```
DATA1       DB    +96
DATA2       DB    +70
RESULT      DW    ?
            ......
            SUB   AH,AH        ;AH=0
            MOV   AL,DATA1     ;GET OPERAND 1
            MOV   BL,DATA2     ;GET OPERAND 2
            ADD   AL,BL        ;ADD THEM
            JNO   OVER         ;IF OF=0 THEN GO TO OVER
            MOV   AL,DATA2     ;OTHERWISE GET OPERAND 2 TO
            CBW                ;SIGN EXTEND IT
            MOV   BX,AX        ;SAVE IT IN BX
            MOV   AL,DATA1     ;GET BACK OPERAND 1 TO
            CBW                ;SIGN EXTEND IT
            ADD   AX,BX        ;ADD THEM AND
OVER:       MOV   RESULT,AX    ;SAVE IT
```

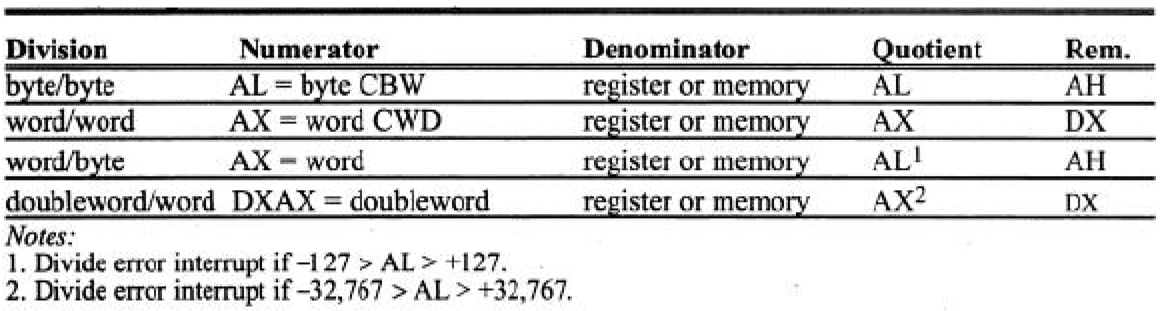


```
S     AH          AL
0     000 0000    0110 0000    +96   after sign extension
0     000 0000    0100 0110    +70   after sign extension
0     000 0000    1010 0110    +166
```

IDIV (signed number division):

The Intel manual says that IDIV means "integer division"; it is used for signed number division. In actuality, all arithmetic instructions of 8088/86 are for integer numbers regardless of whether the operands are signed or unsigned. To perform operations on real numbers, the 8087 coprocessor is used. Remember that real numbers are the ones with decimal points such as "3.56".

Division of signed numbers is very similar to the division of unsigned numbers (already discussed).



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

Notes:
1. Divide error interrupt if –127 > AL > +127.
2. Divide error interrupt if –32,767 > AL > +32,767.

Eg1:

IDIV CH

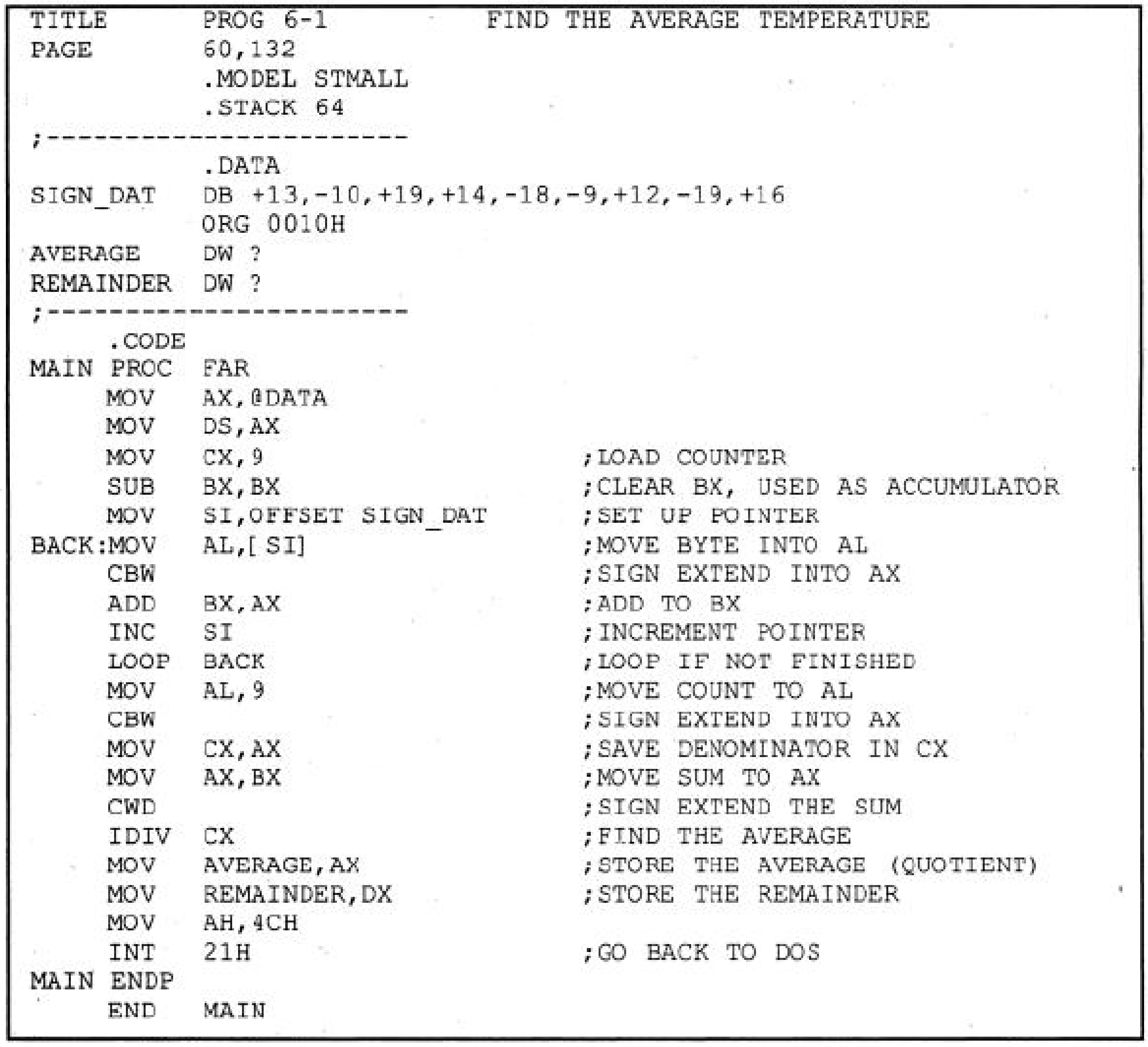
		Before	After	
F0H = -10H	CH	F0H		EE = -12H
	AL	25H	EEH	Quotient
	AH	01H	05H	Remainder

Eg2:

IDIV BL

		Before	After	
F0H = -3H	BL	FDH		FB = -5H
	AL	10H	EBH	Quotient
	AH	00H	01H	Remainder

An application of signed number arithmetic is given in the following Program. It computes the average of the Celsius temperatures: +13, -10, + 19, +14, -18, -9, +12, -19, and + 16.

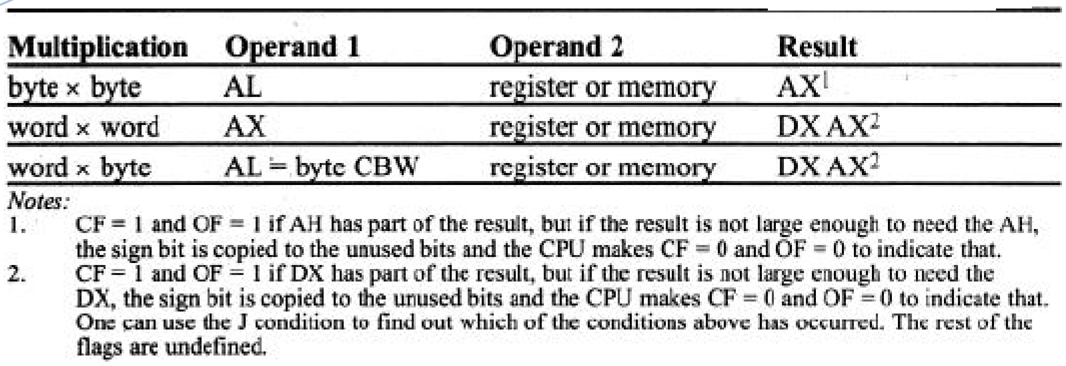


```
TITLE       PROG 6-1          FIND THE AVERAGE TEMPERATURE
PAGE        60,132
            .MODEL STMALL
            .STACK 64
;-----------------------
            .DATA
SIGN_DAT    DB +13,-10,+19,+14,-18,-9,+12,-19,+16
            ORG 0010H
AVERAGE     DW ?
REMAINDER   DW ?
;-----------------------
      .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    CX,9                    ;LOAD COUNTER
     SUB    BX,BX                   ;CLEAR BX, USED AS ACCUMULATOR
     MOV    SI,OFFSET SIGN_DAT      ;SET UP POINTER
BACK:MOV    AL,[SI]                 ;MOVE BYTE INTO AL
     CBW                            ;SIGN EXTEND INTO AX
     ADD    BX,AX                   ;ADD TO BX
     INC    SI                      ;INCREMENT POINTER
     LOOP   BACK                    ;LOOP IF NOT FINISHED
     MOV    AL,9                    ;MOVE COUNT TO AL
     CBW                            ;SIGN EXTEND INTO AX
     MOV    CX,AX                   ;SAVE DENOMINATOR IN CX
     MOV    AX,BX                   ;MOVE SUM TO AX
     CWD                            ;SIGN EXTEND THE SUM
     IDIV   CX                      ;FIND THE AVERAGE
     MOV    AVERAGE,AX              ;STORE THE AVERAGE (QUOTIENT)
     MOV    REMAINDER,DX            ;STORE THE REMAINDER
     MOV    AH,4CH
     INT    21H                     ;GO BACK TO DOS
MAIN ENDP
     END    MAIN
```

Program 6-1

IMUL (signed number multiplication)

Signed number multiplication is similar in its operation to the unsigned multiplication. The only difference between them is that the operands in signed number operations can be positive or negative; therefore, the result must indicate the sign.



Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX[1]
word × word	AX	register or memory	DX AX[2]
word × byte	AL = byte CBW	register or memory	DX AX[2]

Notes:

1. CF = 1 and OF = 1 if AH has part of the result, but if the result is not large enough to need the AH, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that.
2. CF = 1 and OF = 1 if DX has part of the result, but if the result is not large enough to need the DX, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that. One can use the J condition to find out which of the conditions above has occurred. The rest of the flags are undefined.

Eg1:

IMUL CH

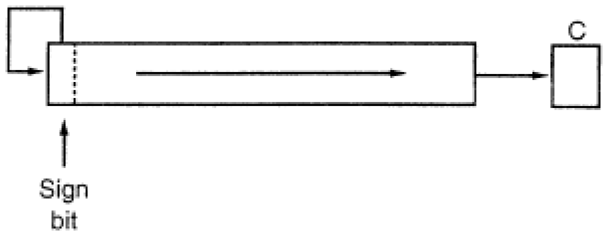
		Before	After	
FEH = -02	CH	FEH		
	AL	02H	FCH	FFFCH = -04
	AH	34H	FFH	

Arithmetic Shift:

The arithmetic shift is used for signed numbers. It is basically the same as the logical shift, except that the sign bit is copied to the shifted bits. SAR (shift arithmetic right) and SAL (shift arithmetic left) are two instructions for the arithmetic shift.

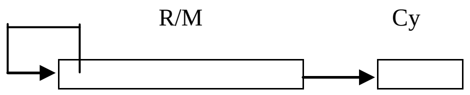
SAR (shift arithmetic right)

```
SAR    destination,count
```



Eg:

SAR BH, CL

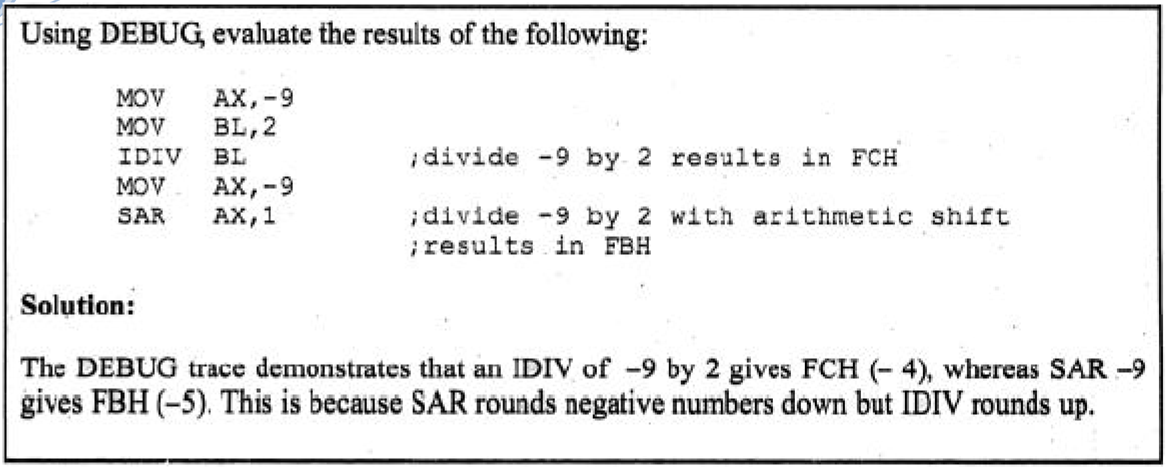


Shift right

		Before	*After*
1100 0000 = -40H	BH	1100 0000	1111 0000
1111 0000 = -10H	CL	02H	
	Cy	1	0

As the bits of the destination are shifted to the right into CF, the empty bits are filled with the sign bit. One can use the SAR instruction to divide a signed number by 2, as shown next:

```
MOV   AL,-10        ;AL=-10=F6H=1111 0110
SAR   AL,1          ;AL is arithmetic shifted right once
                    ;AL=1111 1011=FDH=-5
```



Using DEBUG, evaluate the results of the following:

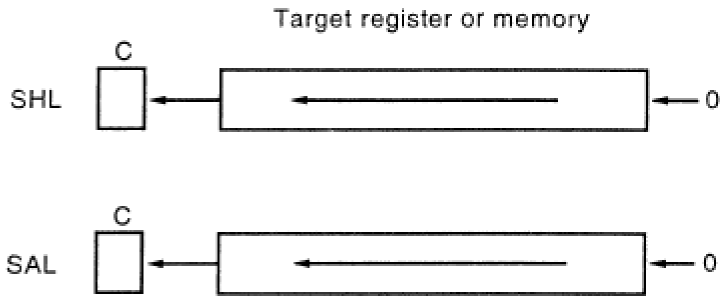
```
MOV   AX,-9
MOV   BL,2
IDIV  BL          ;divide -9 by 2 results in FCH
MOV   AX,-9
SAR   AX,1        ;divide -9 by 2 with arithmetic shift
                  ;results in FBH
```

Solution:

The DEBUG trace demonstrates that an IDIV of –9 by 2 gives FCH (– 4), whereas SAR –9 gives FBH (–5). This is because SAR rounds negative numbers down but IDIV rounds up.

SAL (shift arithmetic left)

SAL & SHL (shift left) do exactly the same thing.



Signed Number Comparison

```
CMP    dest,source
```

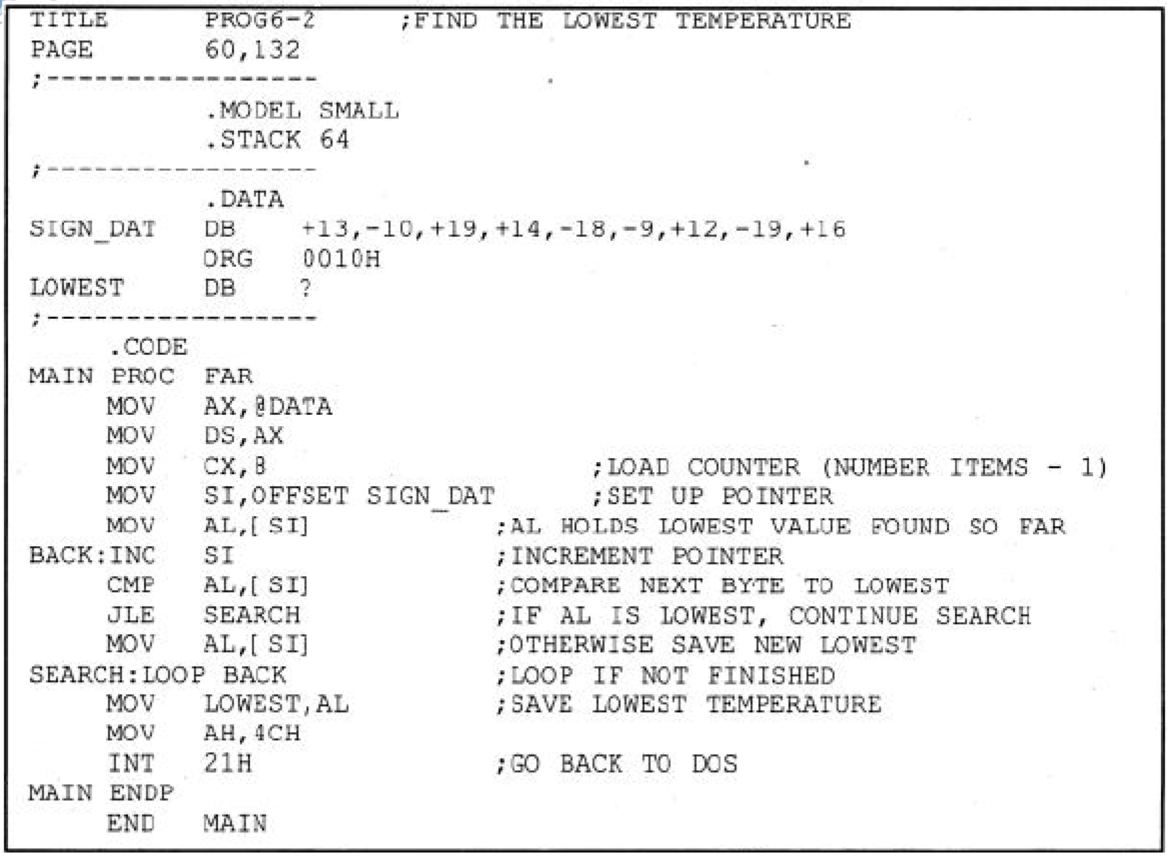
Although the CMP (compare) instruction is the same for both signed and unsigned numbers, the J condition instruction used to make a decision for the signed numbers is different from that used for the unsigned numbers.

- In unsigned number comparisons, CF and ZF are checked for conditions of larger, equal, and smaller.
- In signed number comparison, OF, ZF, and SF are checked.

```
destination > source    OF=SF or ZF=0
destination = source    ZF=1
destination < source    OF=negation of SF
```

- The memories used to detect the conditions above are as follows:

```
JG     Jump Greater            jump if OF=SF or ZF=0
JGE    Jump Greater or Equal   jump if OF=SF
JL     Jump Less               jump if OF=inverse of SF
JLE    Jump Less or Equal      jump if OF=inverse of SF or ZF=1
JE     Jump if Equal           jump of ZF = 1
```



```
TITLE       PROG6-2     ;FIND THE LOWEST TEMPERATURE
PAGE        60,132
;-----------------
            .MODEL SMALL
            .STACK 64
;-----------------
            .DATA
SIGN_DAT    DB    +13,-10,+19,+14,-18,-9,+12,-19,+16
            ORG   0010H
LOWEST      DB    ?
;-----------------
      .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      MOV   CX,8                   ;LOAD COUNTER (NUMBER ITEMS - 1)
      MOV   SI,OFFSET SIGN_DAT     ;SET UP POINTER
      MOV   AL,[SI]          ;AL HOLDS LOWEST VALUE FOUND SO FAR
BACK:INC    SI               ;INCREMENT POINTER
      CMP   AL,[SI]          ;COMPARE NEXT BYTE TO LOWEST
      JLE   SEARCH           ;IF AL IS LOWEST, CONTINUE SEARCH
      MOV   AL,[SI]          ;OTHERWISE SAVE NEW LOWEST
SEARCH:LOOP BACK             ;LOOP IF NOT FINISHED
      MOV   LOWEST,AL        ;SAVE LOWEST TEMPERATURE
      MOV   AH,4CH
      INT   21H              ;GO BACK TO DOS
MAIN  ENDP
      END   MAIN
```

Program 6-2

STRING & TABLE OPERATIONS:

- There is a group of instructions referred to as string instructions in the x86 family of microprocessors.
- They are capable of performing operations on a series of operands located in consecutive memory locations.
- For example, while the CMP instruction can compare only 2 bytes (or words) of data, the CMPS (compare string) instruction is capable of comparing two arrays of data located in memory locations pointed at by the SI and DI registers. These instructions are very powerful and can be used in many applications,

Use of SI and DI, DS and ES in String Instructions:

- For string operations to work, designers of CPUs must set aside certain registers for specific functions. These registers must permanently provide the source and destination operands.
- In 088/86 microprocessor, the SI and DI registers always point to the source and destination operands, respectively.
- To generate the physical address, the 8088/86 always uses SI as the offset of the DS (data segment) register and DI as the offset of ES (extra segment).
- The ES register must be initialized for the string operation(s) to work.

Byte and Word Operands in String Instructions:

- In each of the string instructions, the operand can be a byte or a word.
- Operands are distinguished by the letters B (byte) and W (word) in the instruction mnemonic.

DF, the Direction Flag:

- To process operands located in consecutive memory locations; it requires that, the pointer be incremented or decremented.
- In string operations this is achieved by the direction flag. Of the 16 bits of the flag register (D0 – D15), bit 11 (D10) is set aside for the direction flag (DF).
- It is the job of the string instruction to increment or decrement the SI and DI pointers; but it is the job of the programmer to specify the choice of increment or decrement by setting the direction flag to high or low.
- The instructions CLD (*clear direction flag*) and STD (*set direction flag*) are specifically designed for the purpose.
- CLD (clear direction flag) will reset (put to zero) the DF, indicating that the string instruction should increment the pointers automatically. This is referred to as *auto-increment*.
- STD (set the direction flag) sets DF to 1, indicating to the string instruction that the pointers SI and DI should be decremented automatically. This is referred to as *auto-decrement*.

Table: Summary of String Operations

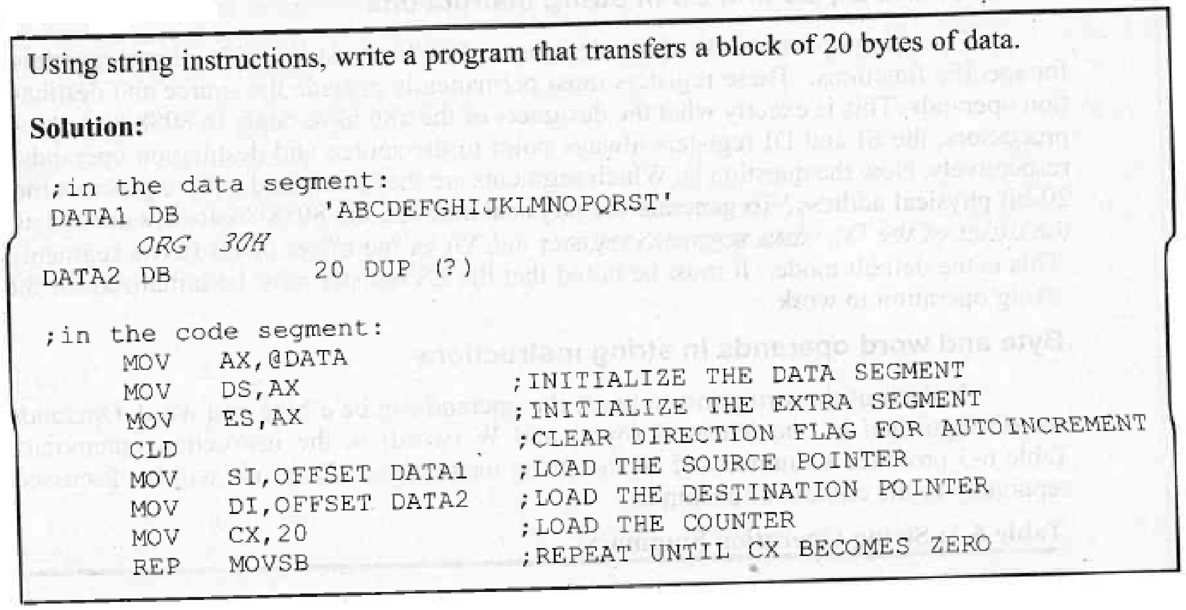
Instruction	Mnemonic	Destination	Source	Prefix
Move string byte	MOVSB	ES: DI	DS: SI	REP
Move string word	MOVSW	ES: DI	DS: SI	REP
Store string byte	STOSB	ES: DI	AL	REP
Store string word	STOSW	ES: DI	AX	REP
Load string byte	LODSB	AL	DS: SI	None
Load string word	LODSW	AX	DS: SI	None
Compare string byte	CMPSB	ES: DI	DS: SI	REPE/REPNE
Compare string word	CMPSW	ES: DI	DS: SI	REPE/REPNE
Scan string byte	SCASB	ES: DI	AL	REPE/REPNE
Scan string word	SCASW	ES: DI	AX	REPE/REPNE

REP/REPZ/REPNZ Prefix:

- **REP (repeat)** prefix allows a string instruction to perform the operation repeatedly.
- REP assumes that CX holds the number of times that the instruction should be repeated.
- In other words, the REP prefix tells the CPU to perform the string operation and then decrements the CX register automatically. This process is repeated until CX becomes zero.

- **REPZ (repeat zero)/REPE (repeat equal)** repeat the string operation as long as source and destination operands are equal (ZF = 1) or until CX becomes zero.
- **REPNZ (repeat not zero)/REPNE (repeat not equal)** repeat the string operation as long as source and destination operands are not equal (ZF = 0) or until CX becomes zero.

Instruction Code	Condition for Exit
REP	CX = 0
REPE/REPZ	CX = 0 or ZF = 0
REPNE/REPNZ	CX = 0 or ZF = 1


Using string instructions, write a program that transfers a block of 20 bytes of data.

Solution:

```
;in the data segment:
DATA1 DB      'ABCDEFGHIJKLMNOPQRST'
      ORG  30H
DATA2 DB      20 DUP (?)

;in the code segment:
      MOV    AX,@DATA
      MOV    DS,AX                ;INITIALIZE THE DATA SEGMENT
      MOV    ES,AX                ;INITIALIZE THE EXTRA SEGMENT
      CLD                         ;CLEAR DIRECTION FLAG FOR AUTOINCREMENT
      MOV    SI,OFFSET DATA1      ;LOAD THE SOURCE POINTER
      MOV    DI,OFFSET DATA2      ;LOAD THE DESTINATION POINTER
      MOV    CX,20                ;LOAD THE COUNTER
      REP    MOVSB                ;REPEAT UNTIL CX BECOMES ZERO
```

- ✓ After the transfer of every byte by the MOVSB instruction, both the SI and DI registers are incremented automatically once only (notice CLD).
- ✓ The REP prefix causes the CX counter to be decremented and MOVSB is repeated until CX becomes zero.
- ✓ An alternative solution for above Example would change only two lines of code:

MOV CX, 10

REP MOVSB

- ✓ In this case the MOVSW will transfer a word (2 bytes) at a time and increment the SI and DI registers each twice. REP will repeat that process until CX becomes zero. Notice that, the CX has the value of 10 in it; since 10 words is equal to 20 bytes.

STOS and LODS Instructions:

STOSB – stores the byte in the AL register into memory location pointed at by ES: DI and then increment DI once (if DF = 0) or decrement DI once (if DF = 1).

STOSW – stores the content of AX in memory locations ES: DI and ES: DI+1 (AL into ES: DI and AH into ES: Dl+1) then increments DI twice (if DF = 0) or decrements DI twice (if DF = 1).

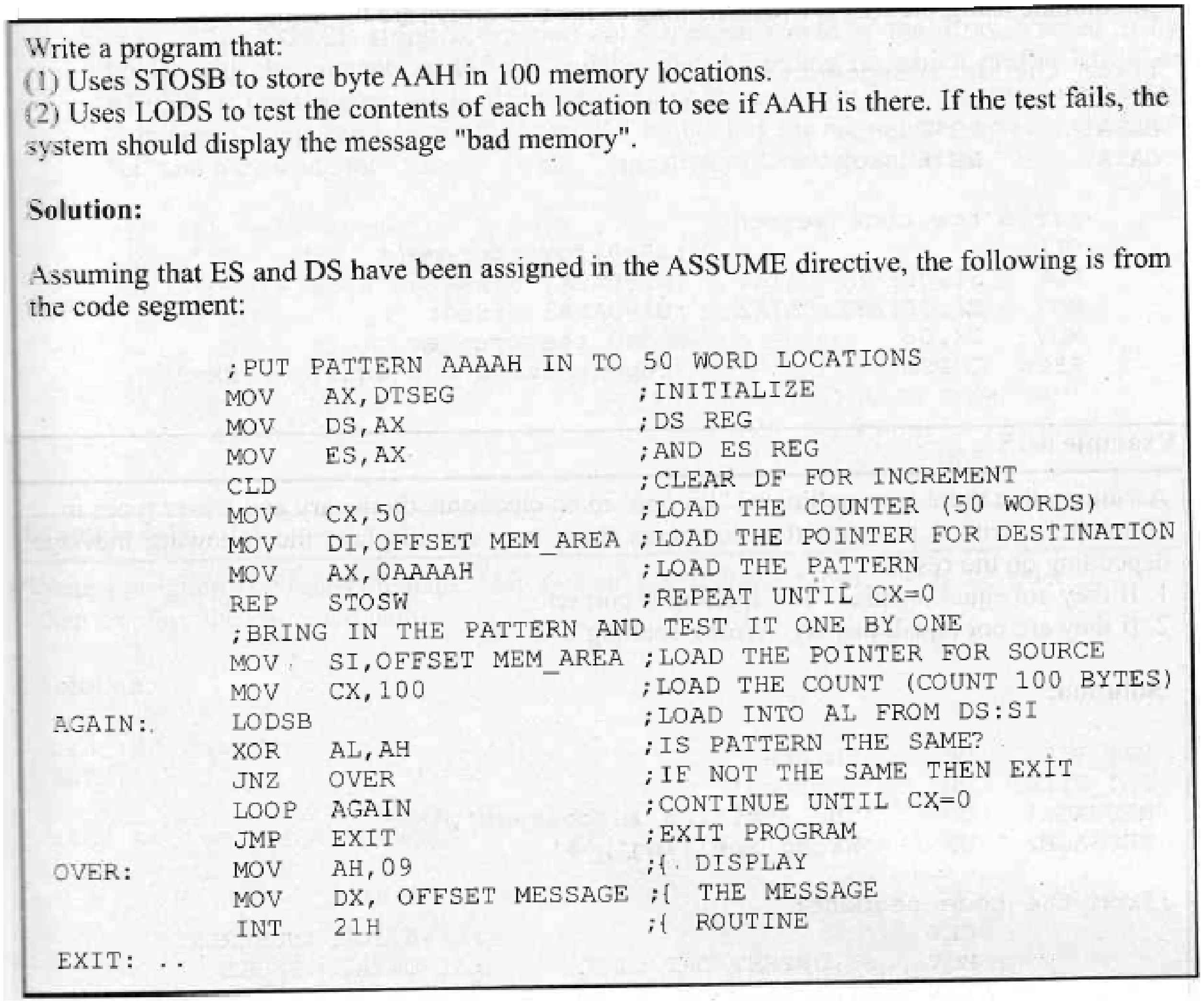
LODSB – loads the contents of memory location pointed at by DS: SI into AL and increments SI once (if DF = 0) or decrements SI once (if DF = l).

LODSW – loads the content of memory locations pointed at by DS: SI into AL and DS: SI+l into AH. The SI is incremented twice if DF = 0 or SI is decremented twice if DF = 1.

- LODS is never used with a REP prefix.

Testing Memory using STOSB and LODSB:

- ✓ The following Example uses string instructions STOSB and LODSB to test an area of RAM memory.
- ✓ First AAH is written into 100 locations by using word-sized operand AAAAH and a count of 50.
- ✓ In the test part, LODSB brings in the contents of memory locations into AL one by one, and each time it is eXclusive-ORed with AAH (the AH register has the hex value of AA).
  - o If they are the same, ZF = l and the process is continued.
  - o Otherwise, the pattern written there by the previous routine is not there and the program will exit.



Write a program that:
(1) Uses STOSB to store byte AAH in 100 memory locations.
(2) Uses LODS to test the contents of each location to see if AAH is there. If the test fails, the system should display the message "bad memory".

Solution:

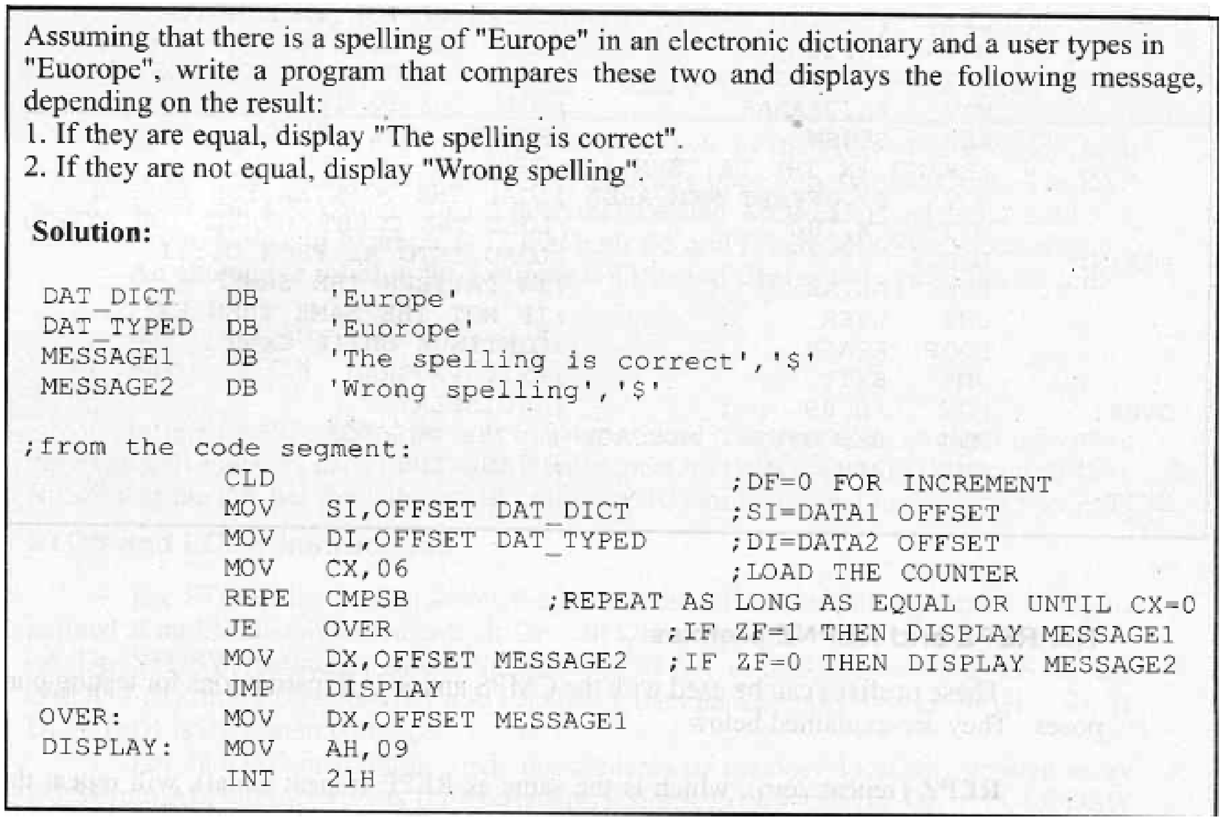
Assuming that ES and DS have been assigned in the ASSUME directive, the following is from the code segment:

```
              ;PUT PATTERN AAAAH IN TO 50 WORD LOCATIONS
              MOV    AX,DTSEG               ;INITIALIZE
              MOV    DS,AX                  ;DS REG
              MOV    ES,AX                  ;AND ES REG
              CLD                           ;CLEAR DF FOR INCREMENT
              MOV    CX,50                  ;LOAD THE COUNTER (50 WORDS)
              MOV    DI,OFFSET MEM_AREA     ;LOAD THE POINTER FOR DESTINATION
              MOV    AX,0AAAAH              ;LOAD THE PATTERN
              REP    STOSW                  ;REPEAT UNTIL CX=0
              ;BRING IN THE PATTERN AND TEST IT ONE BY ONE
              MOV    SI,OFFSET MEM_AREA     ;LOAD THE POINTER FOR SOURCE
              MOV    CX,100                 ;LOAD THE COUNT (COUNT 100 BYTES)
AGAIN:        LODSB                         ;LOAD INTO AL FROM DS:SI
              XOR    AL,AH                  ;IS PATTERN THE SAME?
              JNZ    OVER                   ;IF NOT THE SAME THEN EXIT
              LOOP   AGAIN                  ;CONTINUE UNTIL CX=0
              JMP    EXIT                   ;EXIT PROGRAM
OVER:         MOV    AH,09                  ;{ DISPLAY
              MOV    DX, OFFSET MESSAGE     ;{  THE MESSAGE
              INT    21H                    ;{  ROUTINE
EXIT: ..
```

CMPS (Compare String):

- CMPS allows the comparison of two arrays of data pointed at by the SI and DI registers.
- One can test for the equality or inequality of data by the use of REPE or REPNE prefixes, respectively.
- The comparison can be performed a byte at a time or a word at time by using CMPSB or CMPSW forms of the instruction.

For example, if comparing "Euorop" and "Europe" for equality, the comparison will continue using the REPE CMPS as long as the two arrays are the same.



Assuming that there is a spelling of "Europe" in an electronic dictionary and a user types in "Euorope", write a program that compares these two and displays the following message, depending on the result:
1. If they are equal, display "The spelling is correct".
2. If they are not equal, display "Wrong spelling".

Solution:

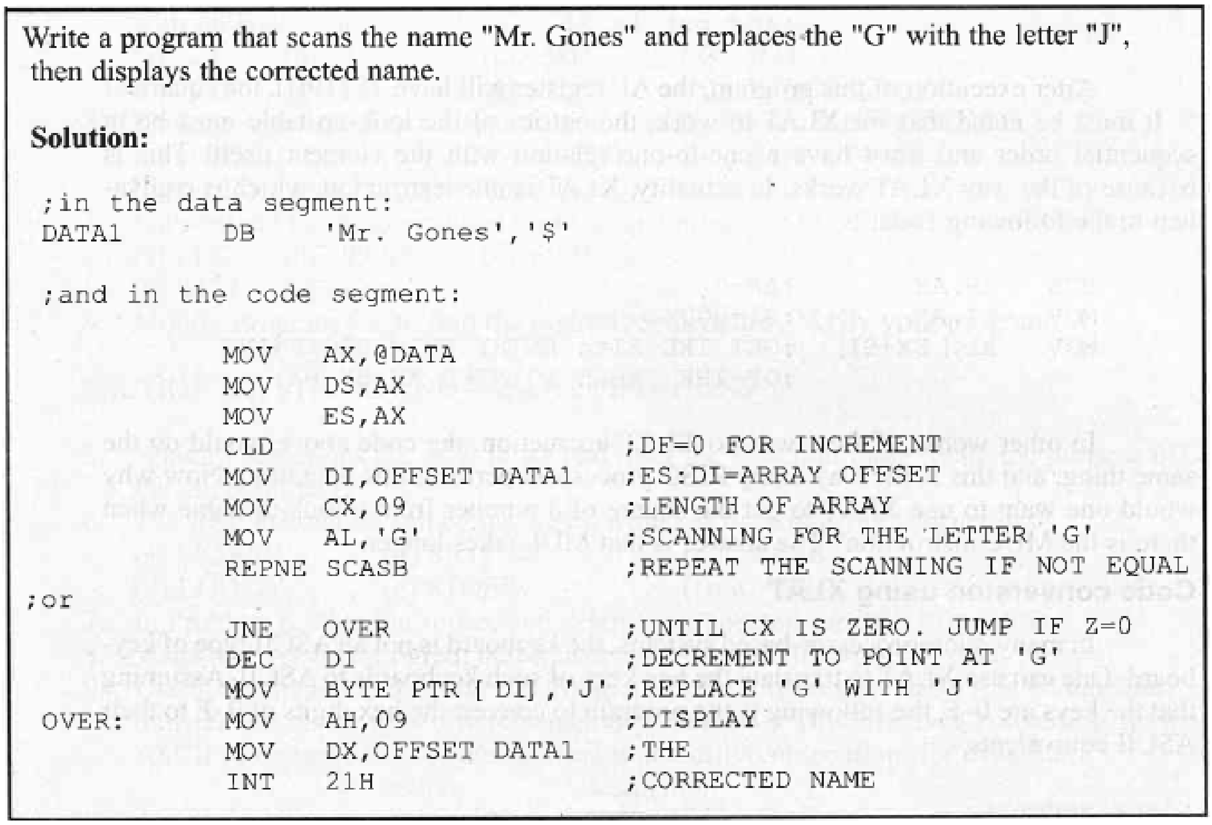
```
 DAT_DICT    DB     'Europe'
 DAT_TYPED   DB     'Euorope'
 MESSAGE1    DB     'The spelling is correct','$'
 MESSAGE2    DB     'Wrong spelling','$'

;from the code segment:
             CLD                                ;DF=0 FOR INCREMENT
             MOV    SI,OFFSET DAT_DICT          ;SI=DATA1 OFFSET
             MOV    DI,OFFSET DAT_TYPED         ;DI=DATA2 OFFSET
             MOV    CX,06                       ;LOAD THE COUNTER
             REPE   CMPSB            ;REPEAT AS LONG AS EQUAL OR UNTIL CX=0
             JE     OVER                    ;IF ZF=1 THEN DISPLAY MESSAGE1
             MOV    DX,OFFSET MESSAGE2      ;IF ZF=0 THEN DISPLAY MESSAGE2
             JMP    DISPLAY
 OVER:       MOV    DX,OFFSET MESSAGE1
 DISPLAY:    MOV    AH,09
             INT    21H
```

- ✓ Here, the two arrays are to be compared letter by letter.
- ✓ The first characters pointed at by SI and DI are compared. In this case they are the same ("E"), so the zero flag is set to 1 and both SI and DI are incremented.
- ✓ Since ZF = 1, the REPE prefix repeats the comparison.
- ✓ This process is repeated until the third letter is reached. The third letters "o" and "r" are not the same; therefore, ZF = 0, and the comparison will stop.

SCAS (Scan String):

- SCASB – compares each byte of the array pointed at by ES: DI with the contents of the AL register, and depending on which prefix, REPE or REPNE, is used, a decision is made for equality or inequality.

- For example, in the array "Mr. Gones", one can scan for the letter "G" by loading the AL register with the character "G" and then using the "REPNE SCASB" operation to look for that letter.



Write a program that scans the name "Mr. Gones" and replaces the "G" with the letter "J", then displays the corrected name.

Solution:

```
;in the data segment:
DATA1      DB     'Mr. Gones','$'

;and in the code segment:
           MOV    AX,@DATA
           MOV    DS,AX
           MOV    ES,AX
           CLD                        ;DF=0 FOR INCREMENT
           MOV    DI,OFFSET DATA1     ;ES:DI=ARRAY OFFSET
           MOV    CX,09               ;LENGTH OF ARRAY
           MOV    AL,'G'              ;SCANNING FOR THE LETTER 'G'
           REPNE SCASB                ;REPEAT THE SCANNING IF NOT EQUAL
;or
           JNE    OVER                ;UNTIL CX IS ZERO. JUMP IF Z=0
           DEC    DI                  ;DECREMENT TO POINT AT 'G'
           MOV    BYTE PTR [DI],'J'   ;REPLACE 'G' WITH 'J'
 OVER:     MOV    AH,09               ;DISPLAY
           MOV    DX,OFFSET DATA1     ;THE
           INT    21H                 ;CORRECTED NAME
```

- ✓ Here, the letter "G" is compared with "M".
- ✓ Since they are not equal, DI is incremented and CX is decremented, and the scanning is repeated until the letter "G" is found or the CX register is zero. In this example, since "G" is found, ZF = 1, indicating that there is a letter "G" in the array.

Replacing the Scanned Character:

- SCASB can be used to search for a character in an array, and if it is found, it will be replaced with the desired character. (See Example given above).
- In string operations the pointer is incremented after each execution (if DF = 0). Therefore, in the example above, DI must be decremented, causing the pointer to point to the scanned character and then replace it.

XLAT Instruction and Look-Up Tables:

- There is often a need in computer applications for a table that holds some important information. To access the elements of the table, 8088/86 microprocessors provide the XLAT (translate) instruction.

- To understand the XLAT instruction, one must first understand tables. The table is commonly referred to as a look-up table.
- Assume that one needs a table for the values of x^2, where x is between 0 and 9. First the table is generated and stored in memory:

```
SQUR_TABLE  DB    0,1,4,9,16,25,36,49,64,81
```

- It is possible to access the square of any number from 0 to 9 by the use of XLAT instruction.
  - ✓ To do that, the register BX must have the offset address of the look-up table, and the number whose square is sought must be in the AL register.
  - ✓ Then after the execution of XLAT, the AL register will have the square of the number.
- The following shows how to get the square of 5 from the table:

```
MOV    BX,OFFSET SQUR_TABLE ;load the offset address of table
MOV    AL,05           ;AL=05 will retrieve 6th element
XLAT                   ;pull the element out of table
                       ;and put in AL
```

- After execution of this program, the AL register will have 25 (19H), the square of 5.
- It must be noted that, for XLAT to work the entries of the look-up table must be in sequential order and must have a one-to-one relation with the element itself. This is because of the way XLAT work.
- In actuality, XLAT is one instruction, which is equivalent to the following code:

```
SUB    AH,AH        ;AH=0
MOV    SI,AX        ;SI=000X
MOV    AL,[BX+SI]   ;GET THE SIth ENTRY FROM BEGINNING
                    ;OF THE TABLE POINTED AT BY BX
```

Code Conversion using XLAT:

- In many microprocessor-based systems, the keyboard is not an ASCII type of keyboard.
- One can use XLAT to translate the hex keys of such keyboards to ASCII.
- Assuming that the keys are 0-F, the following is the program to convert the hex digits of 0-F to their ASCII equivalents.

```
;data segment:
ASC_TABL     DB    '0','1','2','3','4','5','6','7','8'
             DB    '9','A','B','C','D','E','F'
HEX_VALU     DB    ?
ASC_VALU     DB    ?
;code segment:
             MOV   BX,OFFSET ASC_TABL       ;BX= TABLE OFFSET
             MOV   AL,HEX_VALU              ;AL=THE HEX DATA
             XLAT                      ;GET THE ASCII EQUIVALENT
             MOV   ASC_VALU,AL              ;MOVE IT TO MEMORY
```

MEMORY & MEMORY INTERFACING

SEMICONDUCTOR MEMORIES

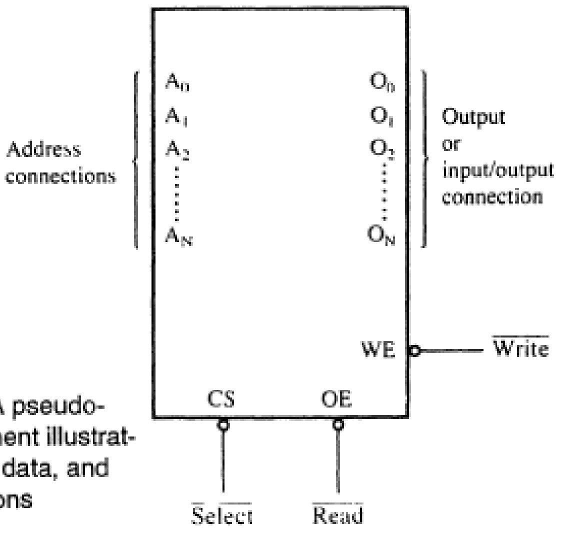
- In the design of computers, semiconductor memories are used as primary storage for code and data. Semiconductor memories are connected directly to the CPU. For this reason, semiconductor memories are referred to as *primary memory*. Most widely used semiconductor memories are ROM and RAM.
- ***Read-only memory (ROM)*** *contains system software and permanent system data.*
- ***Random access memory (RAM)*** or ***read/write memory*** contains temporary data and application software.

Memory Organization:

- The number of bits that a semiconductor memory chip can store is called its *capacity*. It can be in the units of K bits (kilobits)/M bits (megabits).
- Memory chips are organized into a number of locations within the IC. Each location can hold 1 bit, 4-bits, 8-bits, or even 16-bits.
- Each memory chip contains 2^x locations, where x is the number of *address pins* on the chip.
- Each location contains y bits, where y is the number of *data pins* on the chip.
- The entire chip will contain 2^x x y bits – the *capacity* of the chip.

The pin connections common to all memory devices are –

- ***Address Connections.*** All memory devices have address inputs that select a memory location within the memory device. Address inputs are always labeled from A_0 to A_n (Note, 'n' is one less than the total number of address pins). The number of address pins found on a memory device is determined by the number of memory locations found within it.
- ***Data Connections.*** All memory devices have a set of data outputs or input/outputs. The device illustrated in the following Figure has a common set of I/O (input/output) connections.



A pseudo-memory component illustrating the address, data, and control connections

- » As shown in the Fig. above; the memory chips have CS (chip select) pin that must be activated for memory contents to be accessed. That means, no data can be written into or read form the memory chip unless CS is activated.
- » Sometimes, OE (output enable)/RD (read)/WR (write) pins may also be present along with CS pin.

Examples: 1] A given memory chip has 12 address pins and 8 data pins. Find the memory organization and the capacity.

Solution:

- ⇨ Memory chip has 12 address lines ↔ 2^{12} = 4,096 locations.
- ⇨ Memory chip has 8 data lines ↔ Each location hold 8 bits of data.
- ⇨ Thus, the memory organization is 4,096 x 8 = 4K x 8 = 32K bits capacity.

Examples: 2] A 512K memory chip has 8 data pins. Find the organization.

Solution:

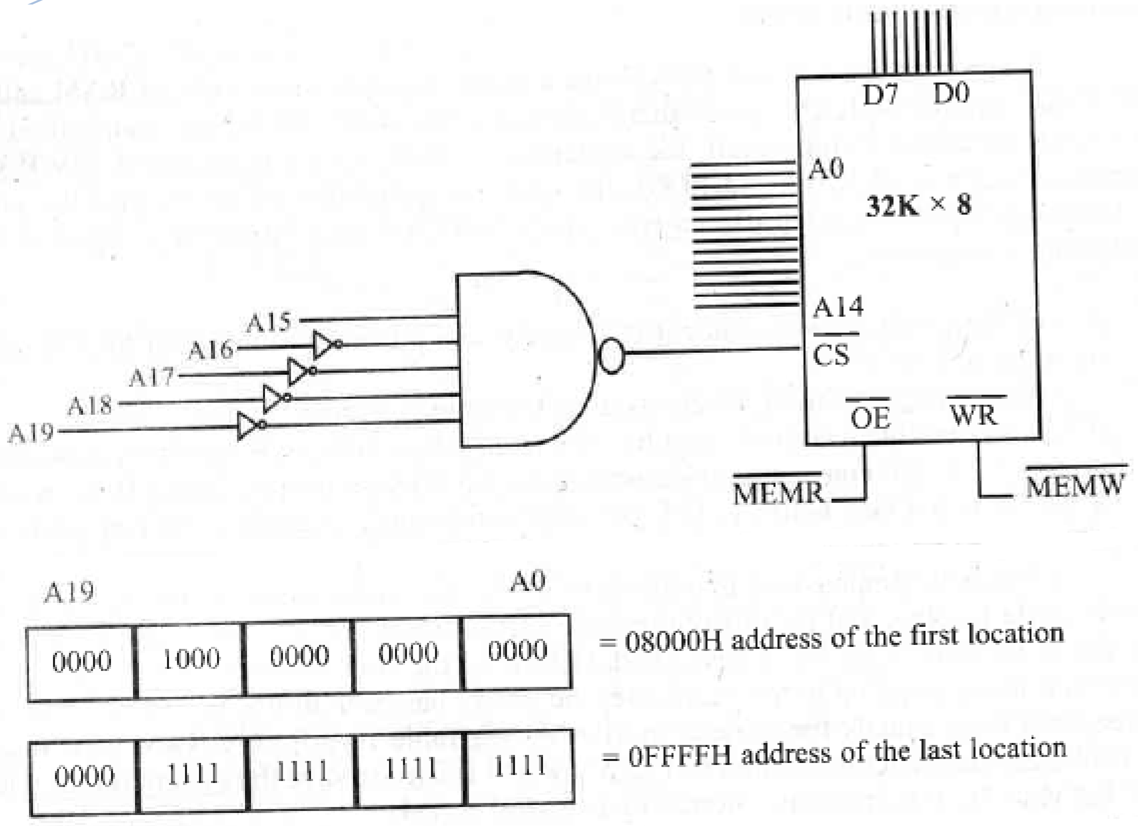
- ⇨ The memory chip has 8 data lines ↔ Each location within the chip can hold 8 bits of data.
- ⇨ Given, the capacity of the memory chip = 512K.
- ⇨ Hence, the locations within the memory chip = 512K / 8 = 64K.
- ⇨ Since, 2^{16} = 64K; the memory chip has 16 address lines.
- ⇨ Hence, the memory organization is: 64K x 8 = 512K bits capacity.

MEMORY ADDRESS DECODING:

- o Consider a 32K x 8 capacity memory chip. This chip has 15 (2^{15} = 32K) address lines and 8 data lines.
- o Suppose, this memory chip is to be interfaced to x86 microprocessor, which is having 20 address lines and 16 data lines.
- o This means that, the microprocessor sends out a 20-bit memory address whenever it reads or writes data. Hence there is a mismatch that must be corrected.
- o The *decoder* corrects the mismatch by decoding the address pins that do not connect to the memory component.

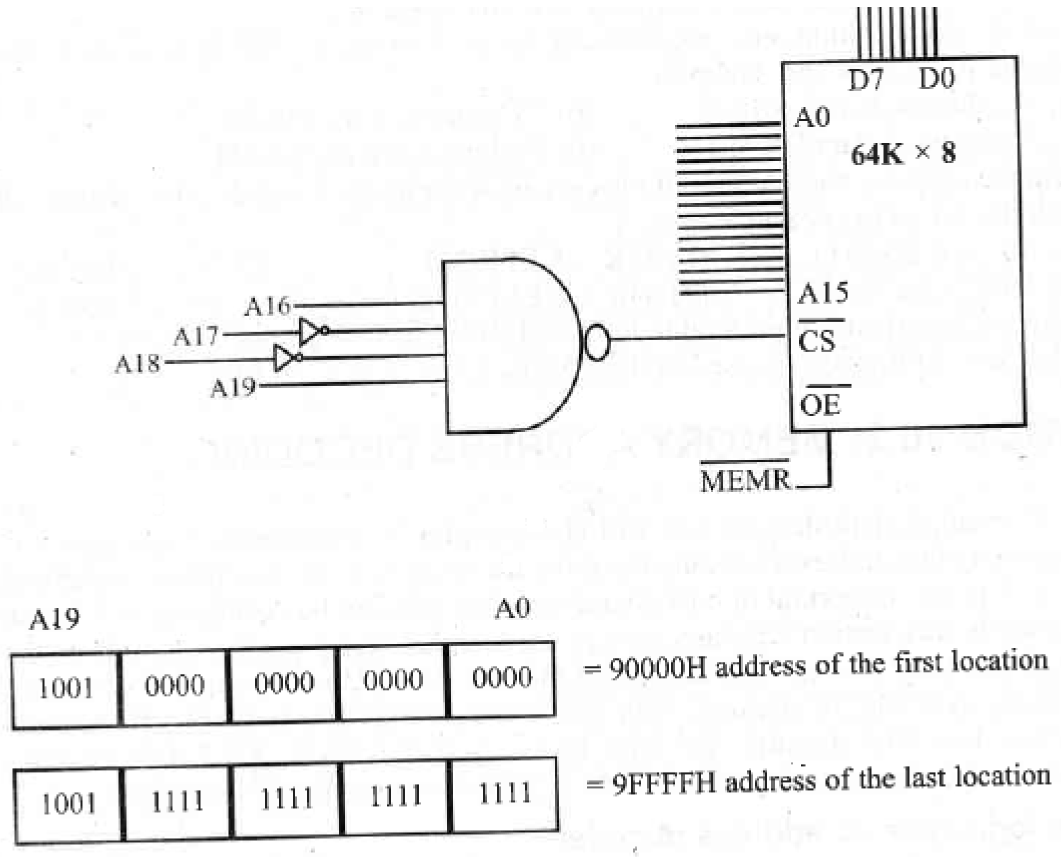
Simple Logic Gates as Address Decoder:

- ✓ The CS (chip select) input pin (in any memory chip) is usually active low and can be activated using some simple logic gates; such as NAND gate and Inverters.
- ✓ The following Fig. shows some simple NAND gate decoding for memory chips, along with the address range calculations.



A19				A0	
0000	1000	0000	0000	0000	= 08000H address of the first location
0000	1111	1111	1111	1111	= 0FFFFH address of the last location

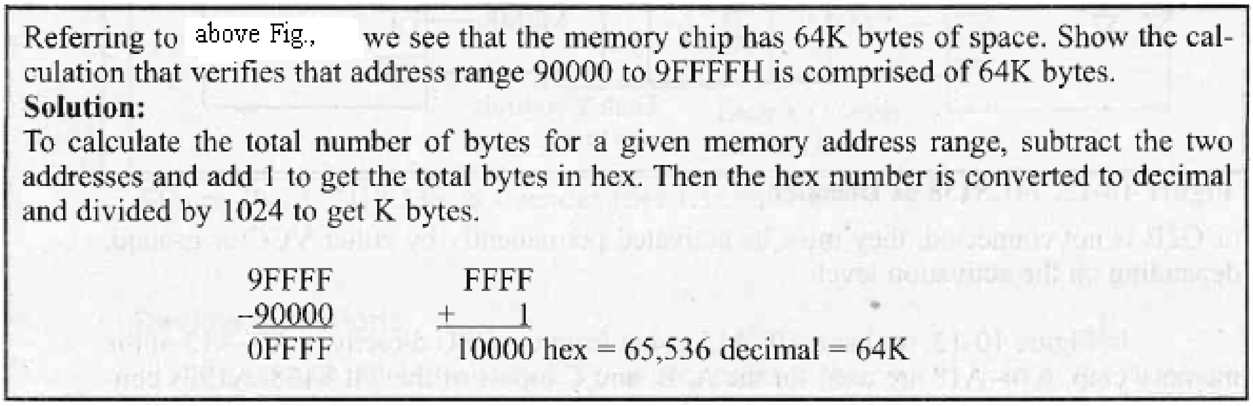
Fig: Simple Logic Gates as Decoder (1)



A19				A0	
1001	0000	0000	0000	0000	= 90000H address of the first location
1001	1111	1111	1111	1111	= 9FFFFH address of the last location

Fig: Simple Logic Gates as Decoder (2)

- Notice that, the output of the NAND gate is active low and that the CS pin is also active low. That makes them a perfect match.
- Also notice that Al9-A16 must equal 1001 in order for CS to be activated. This results in the assignment of addresses 9000H to 9FFFFH to this memory block.



Referring to above Fig., we see that the memory chip has 64K bytes of space. Show the calculation that verifies that address range 90000 to 9FFFFH is comprised of 64K bytes.

Solution:

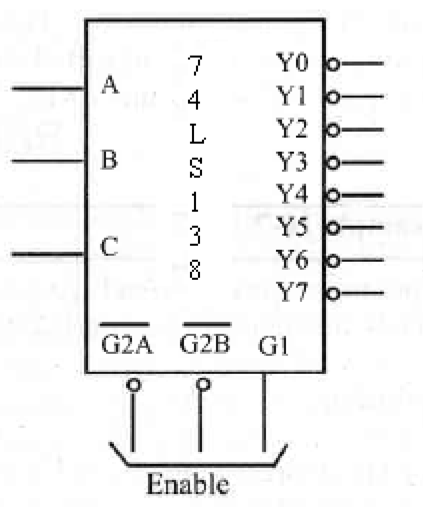
To calculate the total number of bytes for a given memory address range, subtract the two addresses and add 1 to get the total bytes in hex. Then the hex number is converted to decimal and divided by 1024 to get K bytes.

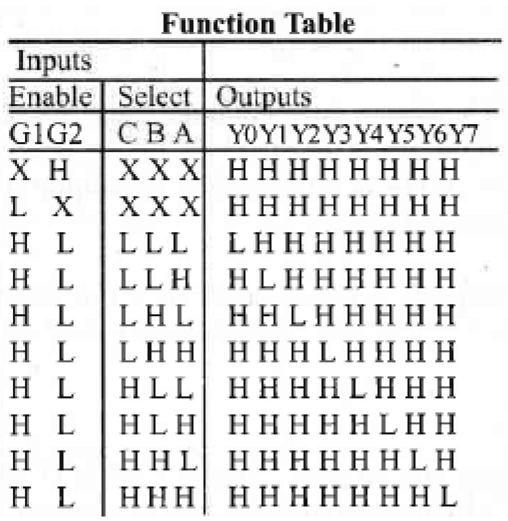
```
 9FFFF         FFFF
-90000       +    1
 0FFFF        10000 hex = 65,536 decimal = 64K
```

Using the 74LS138 as Decoder:

- The 74LS138 has 8 NAND gates in it; therefore, a single chip can control 8 blocks of memory.
- In 74LS138 decoder; the three inputs A, B, C generates eight active low outputs Y0 to Y7.

Block Diagram





Function Table

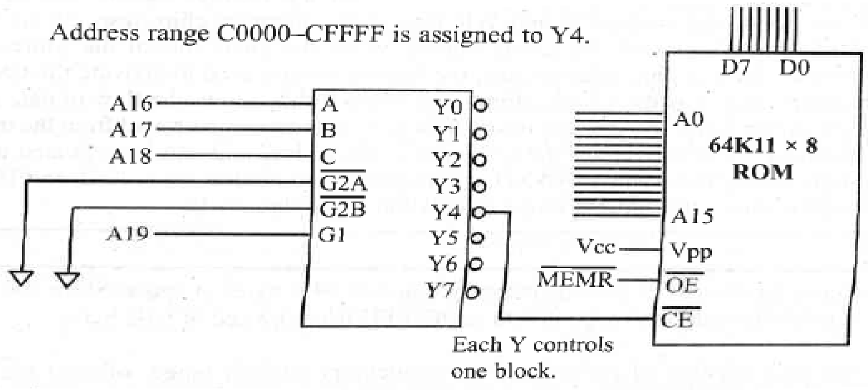
Inputs												
Enable		Select			Outputs							
G1	G2	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

- Each Y output can be connected to the CS of memory chip, allowing control of 8 memory blocks by a single 74LS138.

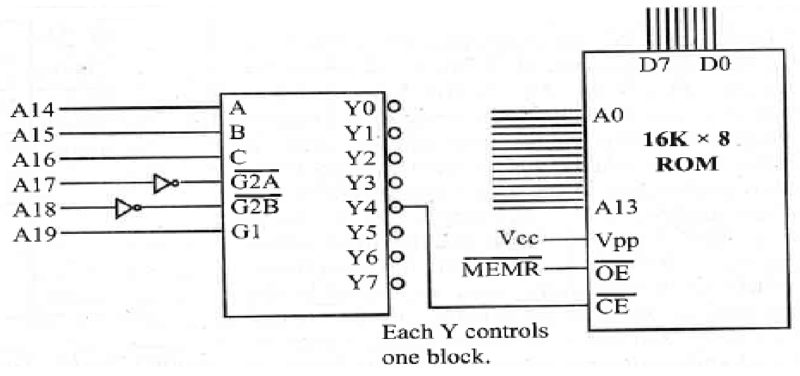
✓ Consider the following memory decoding diagram. We have, A0-A15 from the CPU, directly connected to A0-A15 of the memory chip.

✓ A16-A18 are used for the A, B, and C inputs of 74LS138; A19 is controlling G1 pin. G2A and G2B are grounded.

Address range C0000–CFFFF is assigned to Y4.



- ✓ To enable 74LS138; G2A = 0, G2B = 0; and G1 = 1.
- ✓ To select Y4; CBA = 100.
- ✓ This gives the address range (for the memory chip controlled by Y4): C0000H to CFFFFH.



Looking at the design in above Fig. , find the address range for (a) Y4, (b) Y2, and (c) Y7, and verify the block size controlled by each Y.

Solution:

(a) The address range for Y4 is calculated as follows.

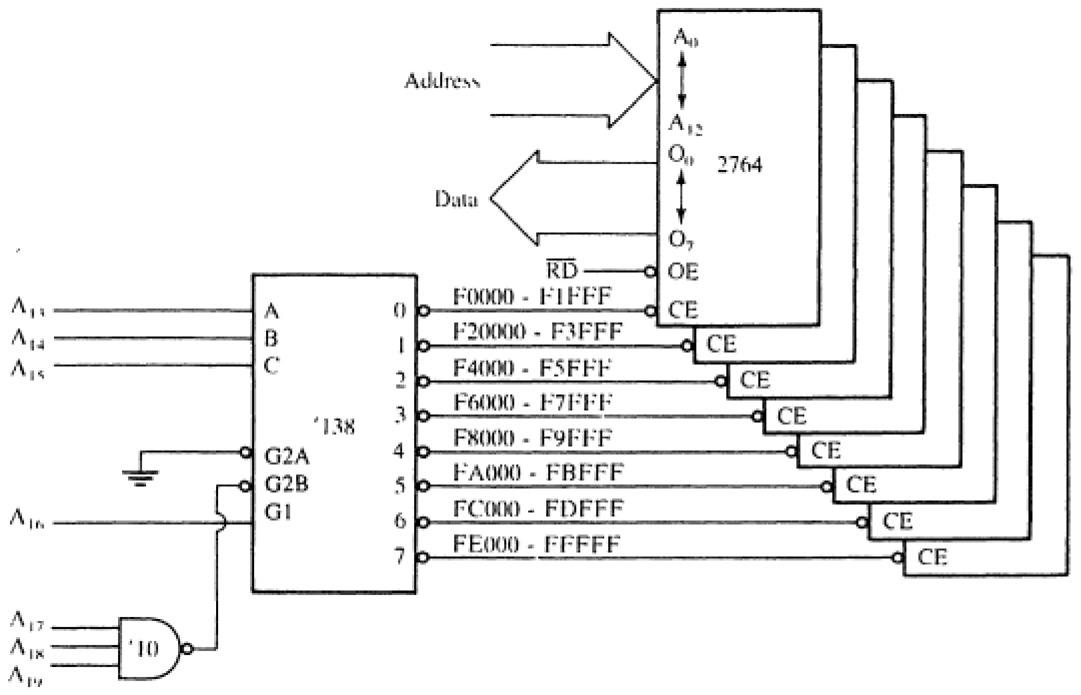
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The above shows that the range for Y4 is F0000H to F3FFFH. In Figure 10-13, notice that A19, A18, and A17 must be 1 for the decoder to be activated. Y4 will be selected when A16 A15 A14 = 100 (4 in binary). The remaining A13–A0 will be 0 for the lowest address and 1 for the highest address.

(b) The address range for Y2 is E8000H to EBFFFH.

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(c) The address range for Y7 is FC000H to FFFFFH. Notice that FFFFF – FC000H = 3FFFH, which is equal to 16,383 in decimal. Adding 1 to it because of the 0 location, we have 16,384. 16,384/1024 = 16K, the block (chip) size.



A circuit that uses eight 2764 EPROMs for a 64K × 8 section of memory in an 8088 microprocessor -based system. The addresses selected in this circuit are F0000H–FFFFFH.

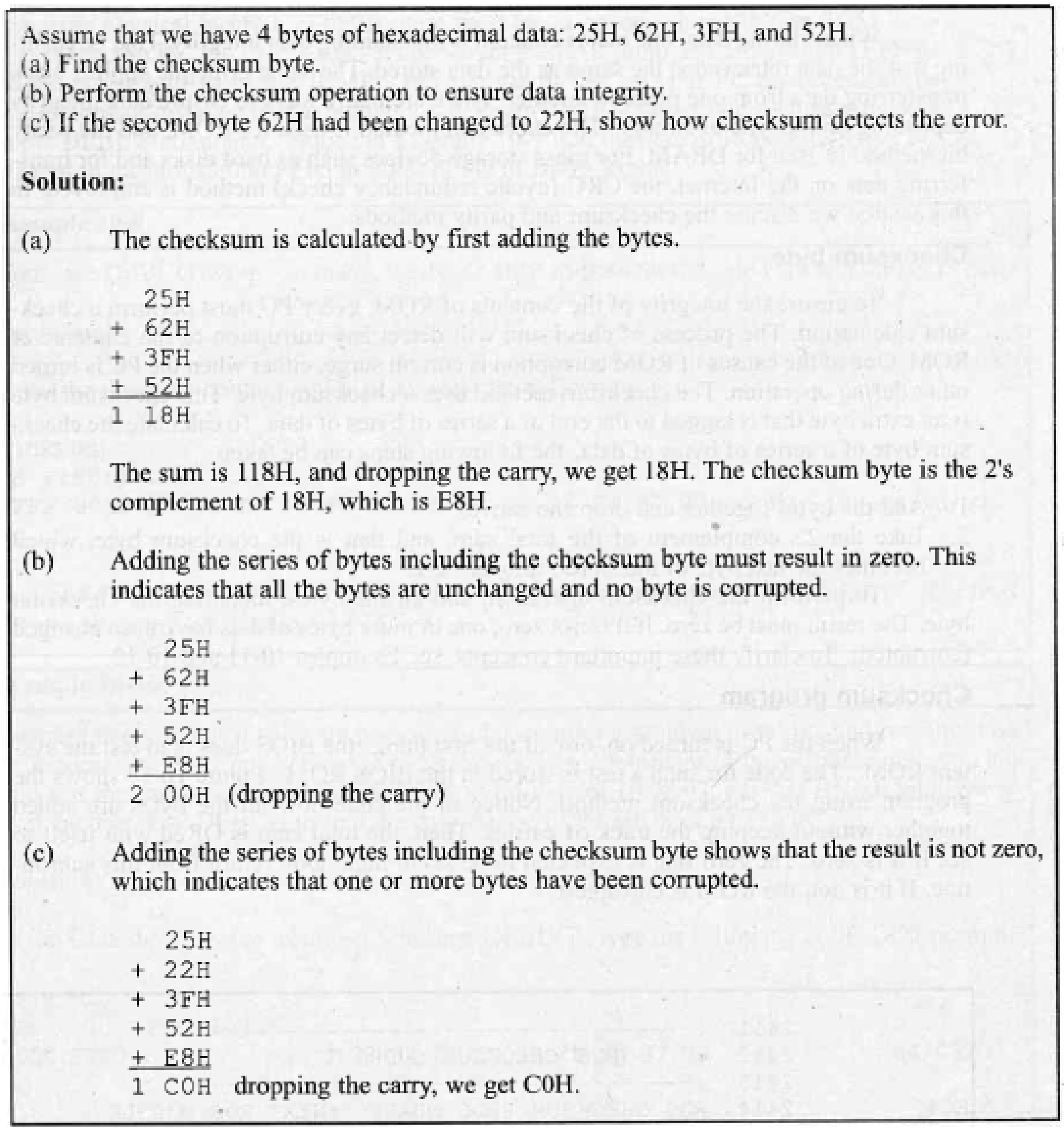
DATA INTEGRITY IN RAM & ROM:

- When storing data, one major concern is maintaining *data integrity – ensuring that, the data retrieved is the same as the data stored.*
- The same principle applies when transferring data from one place to another – *ensuring that, the data received is the same as the data transmitted.*
- There are many way to ensure data integrity depending on the type of storage.
- The *checksum* method is used for ROM and the *parity bit* method is used for DRAM.
- For mass storage devices such as hard disks and for transferring data on the Internet, the *CRC* (*cyclic redundancy check*) method is employed.

Checksum Byte:

- During the current surge, or when the PC is turned on, or during operation, the contents of the ROM may be corrupted.
- To ensure the integrity of the contents of ROM, every PC must perform a checksum calculation. The process of checksum will detect any corruption of the contents of ROM.
- The checksum method uses a checksum byte. This checksum byte is an extra byte that is tagged to the end of a series of bytes of data.
- To calculate the checksum byte of a series of bytes of data, the following steps can be taken .
  1. Add the bytes together and drop the carries.
  2. Take the 2's complement of the total sum, and that is the checksum byte, which becomes the last byte of the stored information.

- To perform the checksum operation, add all the bytes, including the checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted).



Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H.
(a) Find the checksum byte.
(b) Perform the checksum operation to ensure data integrity.
(c) If the second byte 62H had been changed to 22H, show how checksum detects the error.

Solution:

(a) The checksum is calculated by first adding the bytes.

```
   25H
+  62H
+  3FH
+  52H
 1 18H
```

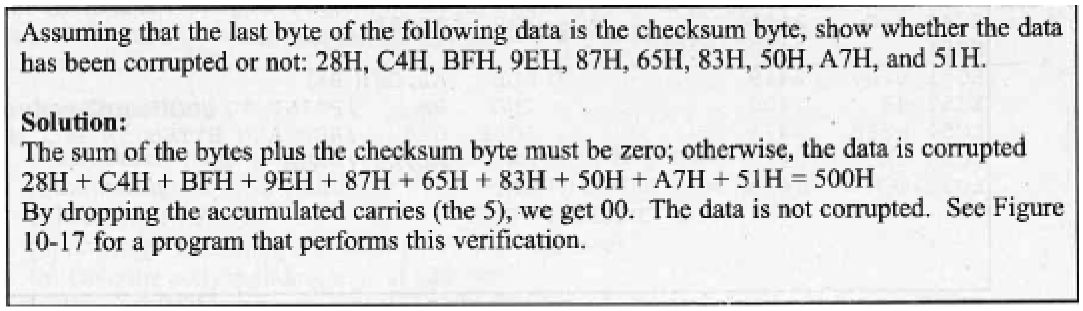
The sum is 118H, and dropping the carry, we get 18H. The checksum byte is the 2's complement of 18H, which is E8H.

(b) Adding the series of bytes including the checksum byte must result in zero. This indicates that all the bytes are unchanged and no byte is corrupted.

```
    25H
 +  62H
 +  3FH
 +  52H
 +  E8H
  2 00H  (dropping the carry)
```

(c) Adding the series of bytes including the checksum byte shows that the result is not zero, which indicates that one or more bytes have been corrupted.

```
    25H
 +  22H
 +  3FH
 +  52H
 +  E8H
  1 C0H   dropping the carry, we get C0H.
```



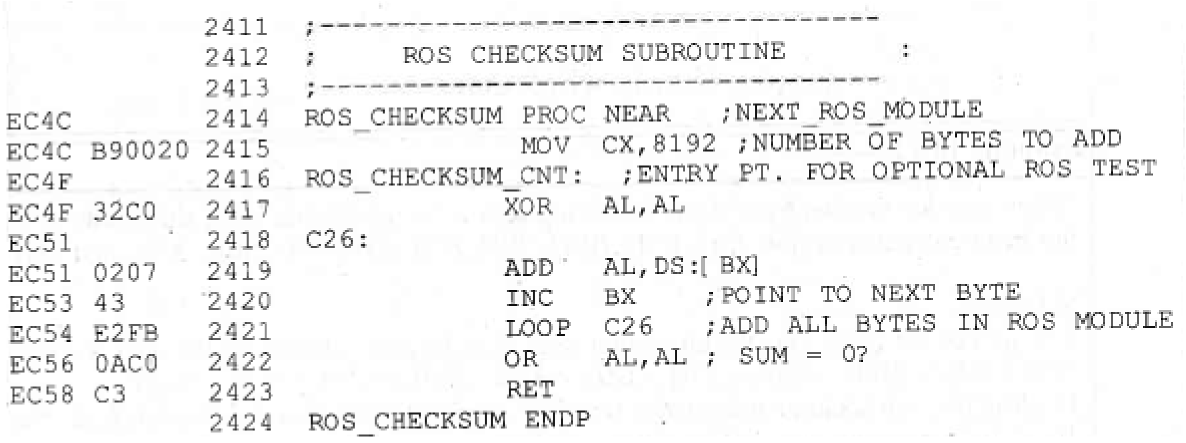
Assuming that the last byte of the following data is the checksum byte, show whether the data has been corrupted or not: 28H, C4H, BFH, 9EH, 87H, 65H, 83H, 50H, A7H, and 51H.

Solution:
The sum of the bytes plus the checksum byte must be zero; otherwise, the data is corrupted
28H + C4H + BFH + 9EH + 87H + 65H + 83H + 50H + A7H + 51H = 500H
By dropping the accumulated carries (the 5), we get 00. The data is not corrupted. See Figure 10-17 for a program that performs this verification.

Checksum Program:

- ✓ When the PC is turned on, one of the first things the BIOS does is to test the system ROM. The code for such a test is stored in the BIOS ROM.
- ✓ The following Figure shows the program using the checksum method.

- ✓ Notice in the code how all the bytes are added together without keeping the track of carries. Then, the total sum is ORed with itself to see if it is zero. The zero flag is expected to be set to high upon return from this subroutine. If it is not, the ROM is corrupted.



```
                ;------------------------------------
                ;     ROS CHECKSUM SUBROUTINE        :
                ;------------------------------------
EC4C            ROS_CHECKSUM PROC NEAR   ;NEXT_ROS_MODULE
EC4C B90020                  MOV  CX,8192 ;NUMBER OF BYTES TO ADD
EC4F            ROS_CHECKSUM_CNT:   ;ENTRY PT. FOR OPTIONAL ROS TEST
EC4F 32C0                    XOR  AL,AL
EC51            C26:
EC51 0207                    ADD  AL,DS:[ BX]
EC53 43                      INC  BX     ;POINT TO NEXT BYTE
EC54 E2FB                    LOOP C26    ;ADD ALL BYTES IN ROS MODULE
EC56 0AC0                    OR   AL,AL ; SUM = 0?
EC58 C3                      RET
                ROS_CHECKSUM ENDP
```

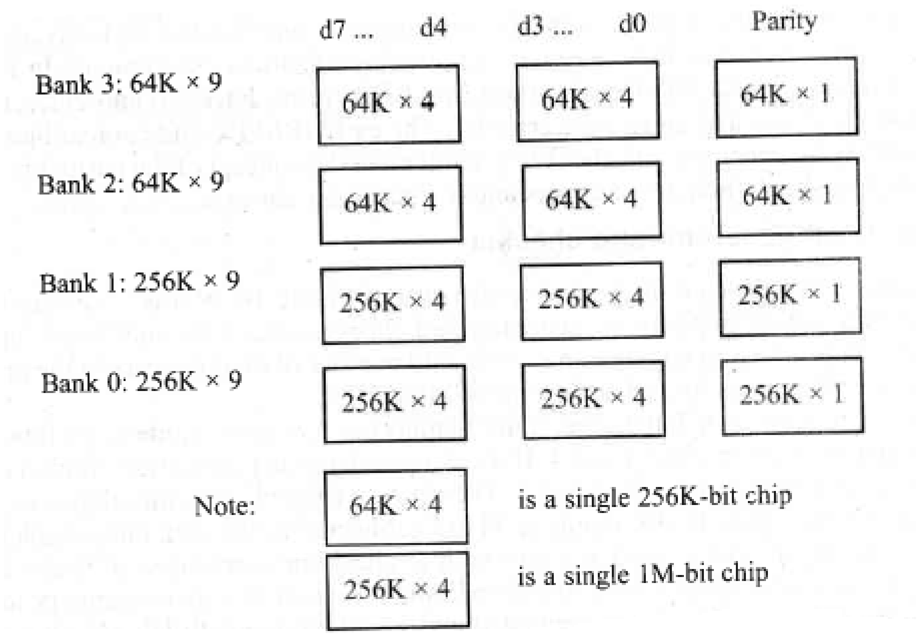
Fig: PC BIOS Checksum Routine

Use of Parity Bit in DRAM Error Detection:

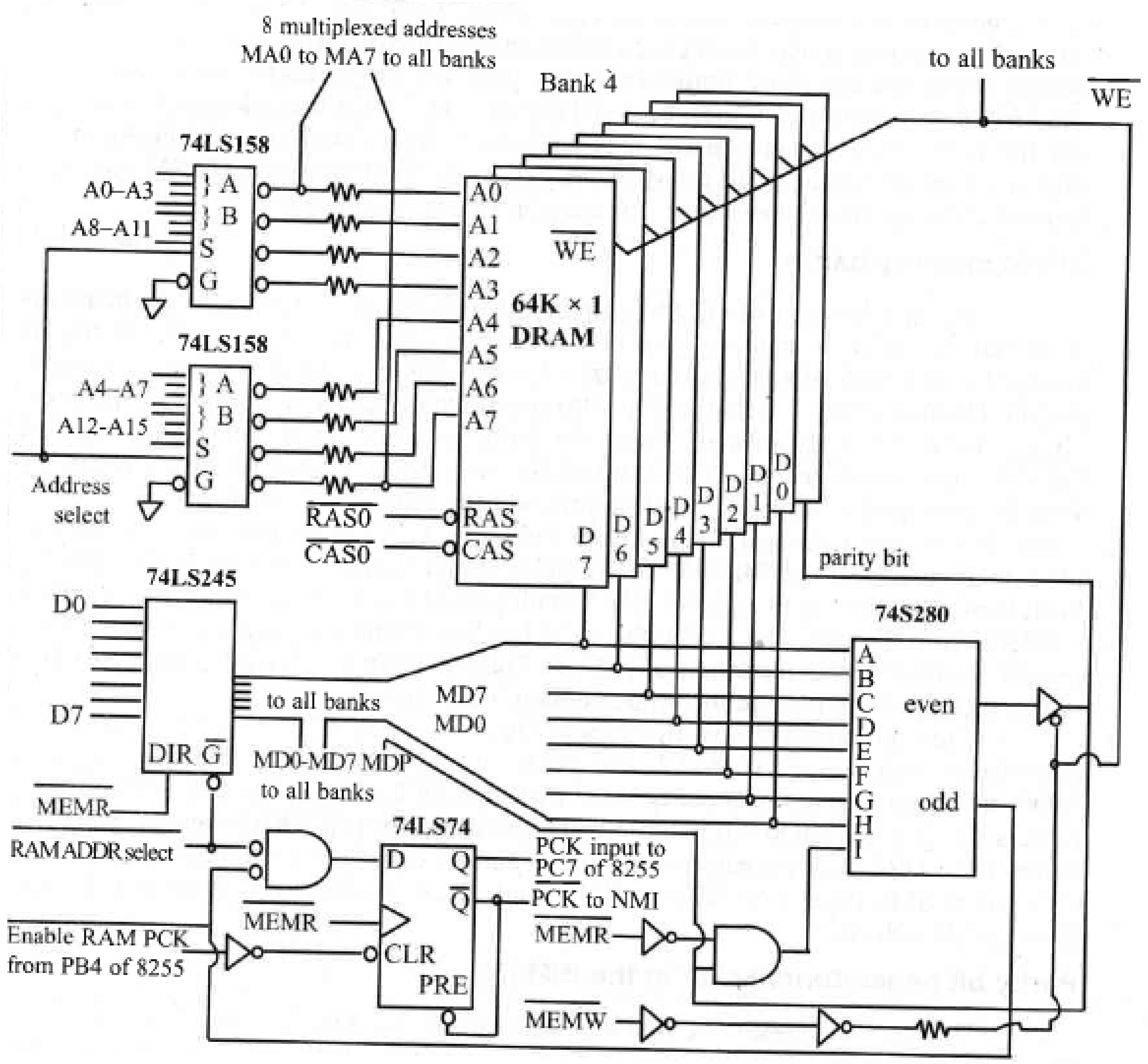
- o System boards or memory modules are populated with DRAM chips of various organizations, depending on the time they were designed and the availability of a given chip at a reasonable cost.
- o The memory technology is changing so fast that DRAM chips on the boards have a different look every year or two. While early PCs used 64K DRAMs, current PCs commonly use 1G chips.
- o To understand the use of a parity bit in detecting data storage errors, we use some simple examples from the early PCs to clarify some very important design concepts.

DRAM Memory Banks:

- ✓ The arrangement of DRAM chips on the system or memory module board is often referred to as a *memory bank*. For example, the 64K bytes of DRAM can be arranged as one bank of 8 IC chips of 64K x 1 organization, or 4 bank of 16K x 1 organization.
- ✓ The first IBM PC introduced in 1981, used memory chip of l6K x l organization.
- ✓ The following Figure shows the memory banks for 640K bytes of RAM using 256K and 1M DRAM chips.
- ✓ Notice the use of an extra bit for every byte of data to store the parity bit.
- ✓ With the extra parity bit every bank requires an extra chip of x 1 organization for parity check.



✓ The following Figure shows DRAM design and parity bit circuitry for a bank of DRAM.



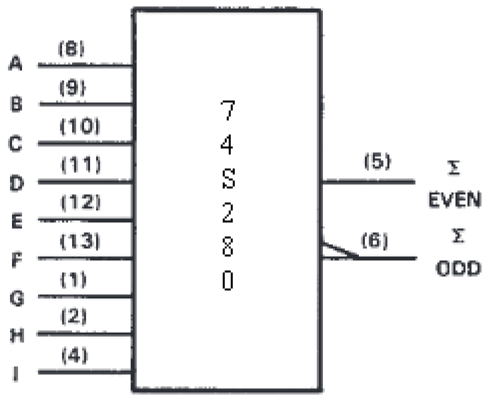
- ✓ First, note the use of the 74LS158 to multiplex the 16 address lines A0-A15, changing them to the 8 address lines of MA0-MA7 (multiplexed address) as required by the 64K x l DRAM chip.
- ✓ The resistors are for the serial bus line termination to prevent undershooting and overshooting at the inputs of DRAM. They range from 20 to 50 ohms, depending on the speed of the CPU and the printed circuit board layout.
- ✓ A few additional observations above Figure should be made. The output of multiplexer addresses MA0-MA7 will go to all the banks. Likewise, memory data MD0-MD7 and memory data parity MDP will go to all the banks.
- ✓ The 74LS245 not only buffers the data bus MD0-MD7 but also boosts it to drive all DRAM inputs. Since the banks of the DRAMs are connected in parallel and the capacitance loading is additive, the data line must be capable of driving all the loads.

Parity Bit Generator/Checker in IBM PC:

- o There are two types of errors that can occur in DRAM chips:
- o *Hard error* – some bits or an entire row of memory cell inside the memory chip get stuck to high or low permanently, thereafter always producing l or 0 regardless of what you write into the cell(s).
- o *Soft error* – a single bit is changed from 1 to 0 or from 0 to 1 due to current surge or certain kinds of particle radiation in the air. Parity is used to detect soft errors.
- o Including a parity bit to ensure data integrity in RAM is the most widely used method; since, it is the simplest and cheapest.
- o This method can only indicate if there is a difference between the data that was written to memory and the data that was read.
- o It cannot correct the error as is the case with some high-performance computers. In those computers and some of the x86-based servers, the EDC (error detection and correction) method is used to detect and correct the error bit.
- o The early IBM PC and compatibles use the 74S280 parity bit generator and checker to implement the concept of the parity bit.

74S280 Parity Bit Generator & Checker:

- ✓ The 74S280 chip has 9 inputs and 2 outputs. Depending on whether an even or odd number of ones appear in the input, the even or odd output is activated (according to following Table).
- ✓ As can be seen from Table, if all 9 inputs have an even number of 1 bits, the even output goes high (as in cases 1 and 4). If the 9 inputs have an odd number of high bits, the odd output goes high (as in cases 2 and 3).



Case	Inputs		Outputs	
	A – H	I	Even	ODD
1	Even	0	1	0
2	Even	1	0	1
3	Odd	0	0	1
4	Odd	1	1	0

The way the IBM PC uses this chip is as follows:

- ✓ Notice that in above Figure (DRAM design and parity bit circuitry for a bank of DRAM), inputs A – H are connected to the data bus, which is 8 bits, or one byte. The I input is used as a parity bit to check the correctness of the byte of data read from memory. When a byte of information is written to a given memory location in DRAM, the even-parity bit is generated and saved on the ninth DRAM chip as a parity bit with use of control signal $\overline{MEMW}$. This is done by activating the tri-state buffer using $\overline{MEMW}$. At this point, I of the 74S280 is equal to zero, since $\overline{MEMR}$ high.
- ✓ When a byte of data is read from the same location, the parity bit is gated into the I input of the 74S280 through $\overline{MEMR}$. This time the odd output is taken out and fed into a 74LS74. If there is a difference between the data written and the data read, the Q output (called PCK, parity bit check) of the 74LS74 is activated and Q activates NMI, indicating that there is a parity bit error, meaning that the data read is not the same as the data written. Consequently, it will display a parity bit error message.
- ✓ For example, if the byte of data written to a location has an even number of ls, A to H has an even number of ls, and I is zero, then the even-parity output of 74S280 becomes 1 and is saved on parity bit DRAM. This is case 1 shown in the above Table. If the same byte of data is read and there is an even number of ls (the byte is unchanged), I from the ninth bit DRAM, which is 1, is input to the 74S280, even becomes low, and odd becomes high, which is case 2 in the above Table. This high from the odd output will be inverted and fed to the 74LS74, making Q low. This means that $\bar{Q}$ is high thereby indicating that the written byte is the same as the byte read and there is no errors occurred.
- ✓ If the number of 1s in the byte has changed from even to odd and the 1 from the saved parity DRAM makes the number of inputs even (case 4 above), the odd output becomes low, which is inverted and passed to the 74LS74 D flip-flop. This makes Q = 1 and $\bar{Q}$ = 0, which signals the NMI to display a parity bit error message on the screen.

16-BIT MEMORY INTERFACING:

In this section, memory interfacing for 16-bit CPUs will be discussed. 80286 is taken as an example, but the concepts can apply to any 16-bit microprocessor.

ODD & EVEN Banks:

In a 16-bit CPU such as the 80286, memory locations 00000-FFFFF are designated as odd and even bytes as shown in the following Fig. This Figure shows only 1M byte of memory; the concept of odd and even banks applies to the entire memory space of a given processor with a 16-bit data bus.

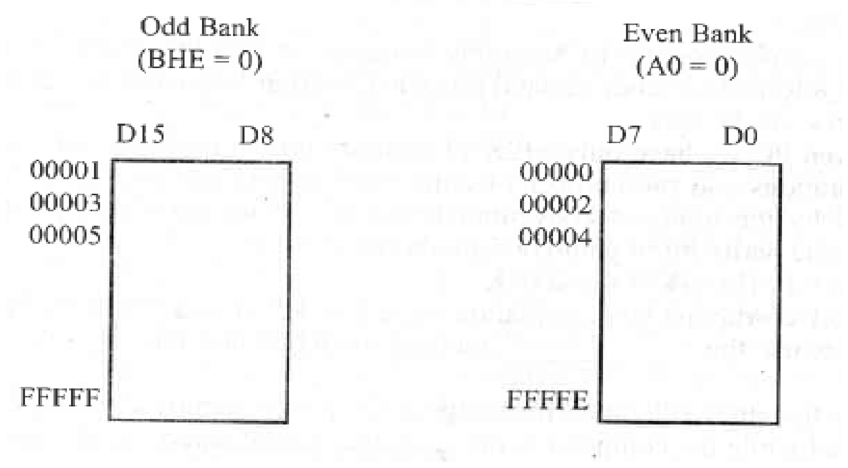


Fig: ODD & EVEN Banks of Memory

To distinguish between odd and even bytes, the CPU provides a signal called BHE (bus high enable). BHE in association with A0 is used to select the odd or even byte according to following Table.

BHE	A0	Memory Selection	
0	0	Even Word	D0 – D15
0	1	Odd Byte	D8 – D15
1	0	Even Byte	D0 – D7
1	1	None	-

The following Figure shows 640KB of DRAM for 16-bit buses.

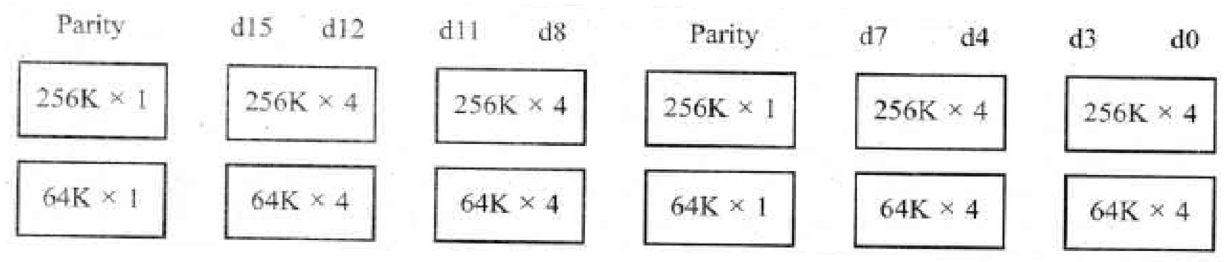


Fig: 640K Bytes of DRAM with ODD & EVEN Banks Designation

The following Figure shows the use of A0 and BHE as bank selectors. Here, the 74LS245 chip is used as a data bus buffer.

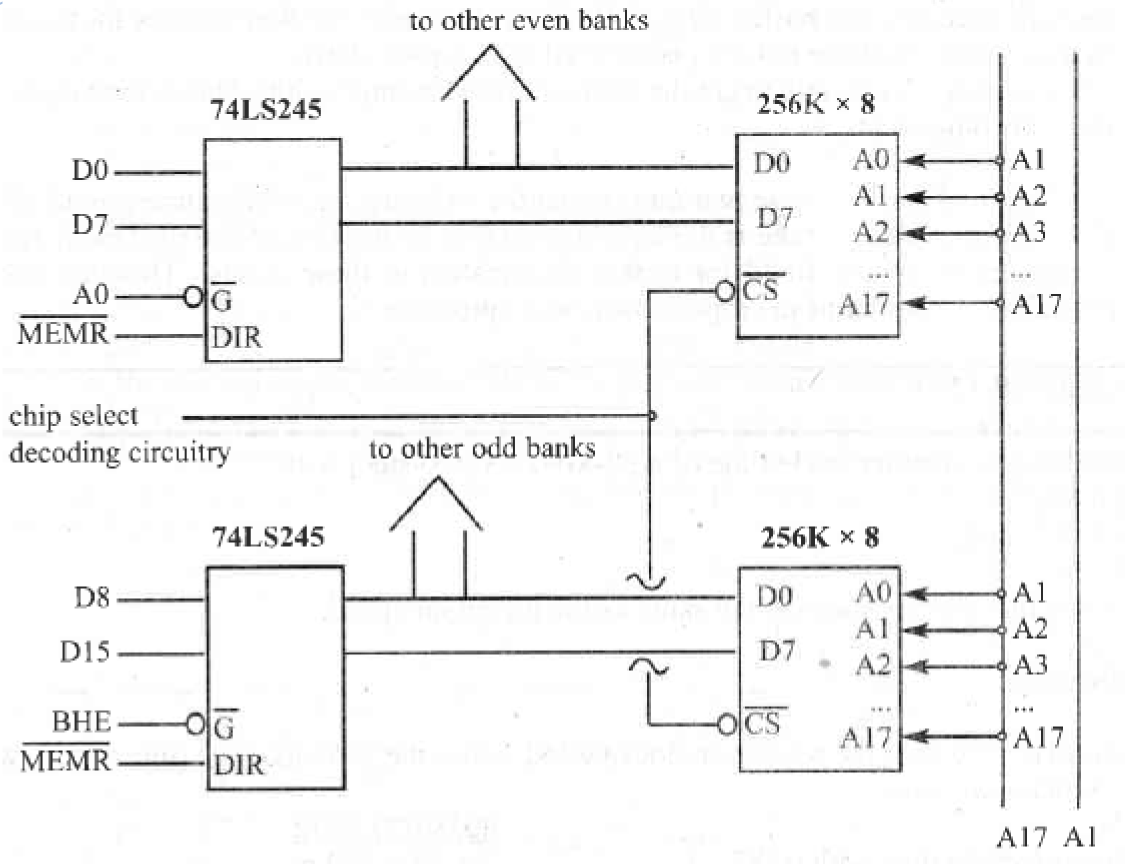
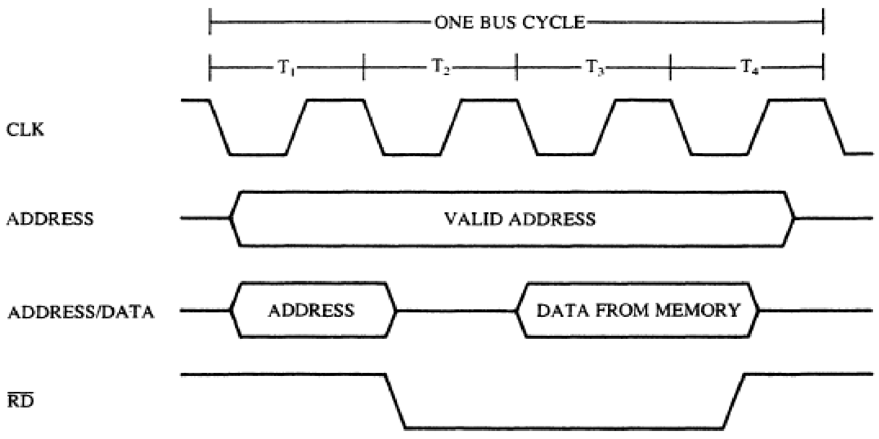


Fig: 16-bit Data Connection in the Systems with 16-bit Data Bus

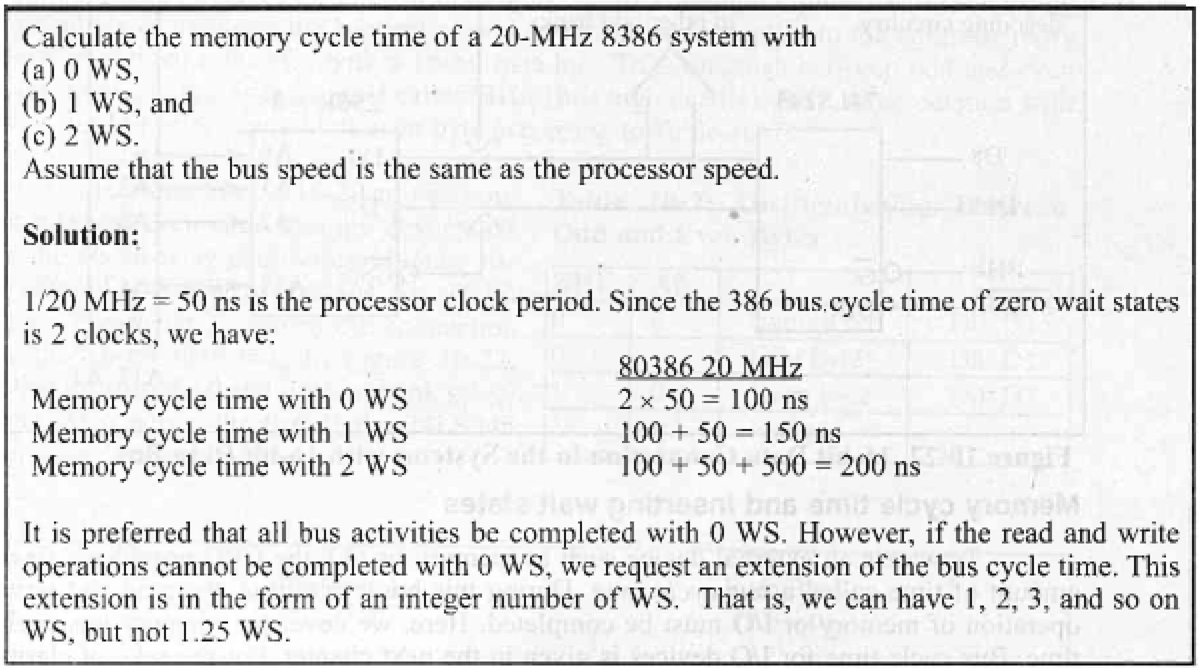
Memory Cycle Time and Inserting Wait States:

- To access an external device such as memory or I/O, the CPU provides a fixed amount of time called a bus cycle time. During this bus cycle time, the read and write operation of memory or I/O must be completed.
- The bus cycle time used for accessing memory is often referred to as MC (memory cycle) time. The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called memory read cycle time.
- The processors such as the 8088/86, the memory cycle time takes 4 clocks, and from 286 to Pentium, the memory cycle time is only 2 clocks.
- If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time. This extra time is called a wait state (WS).



Simplified 8086/8088 read bus cycle

- It must be noted that, memory access time is not the only factor in slowing down the CPU. The other factor is the delay associated with signals going through the data and address path.
- Delay associated with reading data stored in memory has the following two components:
  1. The time taken for address signals to go from CPU pins to memory pins, (going through decoders and buffers (e.g., 74LS245)); plus the time taken for the data to travel from memory to CPU, is referred to as a *path delay*.
  2. The *memory access time* to get the data out of the memory chip. This is the larger (80% of the read cycle time) of the two components.
- The total sum of these two (path delay + memory access time) must equal the memory read cycle time provided by the CPU.



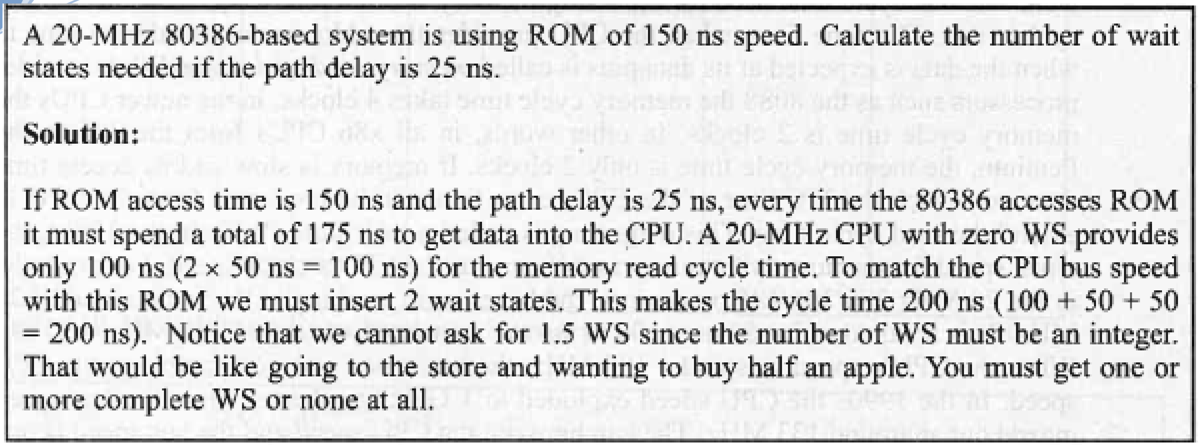
Calculate the memory cycle time of a 20-MHz 8386 system with
(a) 0 WS,
(b) 1 WS, and
(c) 2 WS.
Assume that the bus speed is the same as the processor speed.

Solution:

1/20 MHz = 50 ns is the processor clock period. Since the 386 bus cycle time of zero wait states is 2 clocks, we have:

	<u>80386 20 MHz</u>
Memory cycle time with 0 WS	2 × 50 = 100 ns
Memory cycle time with 1 WS	100 + 50 = 150 ns
Memory cycle time with 2 WS	100 + 50 + 500 = 200 ns

It is preferred that all bus activities be completed with 0 WS. However, if the read and write operations cannot be completed with 0 WS, we request an extension of the bus cycle time. This extension is in the form of an integer number of WS. That is, we can have 1, 2, 3, and so on WS, but not 1.25 WS.



A 20-MHz 80386-based system is using ROM of 150 ns speed. Calculate the number of wait states needed if the path delay is 25 ns.

Solution:

If ROM access time is 150 ns and the path delay is 25 ns, every time the 80386 accesses ROM it must spend a total of 175 ns to get data into the CPU. A 20-MHz CPU with zero WS provides only 100 ns (2 × 50 ns = 100 ns) for the memory read cycle time. To match the CPU bus speed with this ROM we must insert 2 wait states. This makes the cycle time 200 ns (100 + 50 + 50 = 200 ns). Notice that we cannot ask for 1.5 WS since the number of WS must be an integer. That would be like going to the store and wanting to buy half an apple. You must get one or more complete WS or none at all.

Accessing EVEN & ODD Words:

- Intel defines 16-bit data as a word. The address of a word can start at an even or an odd number.
- For example, in the instruction "*MOV AX, [2000]*" the address of the word being fetched into AX starts at an even address. In the case of "*MOV AX, [2007]*" the address starts at an odd address.
- In systems with a 16-bit data bus, accessing a word from an odd addressed location can be slower.
- As shown in the following Fig, in the 8-bit system, accessing a word is treated like accessing two bytes regardless of whether the address is odd or even. Since accessing a byte takes one memory cycle, accessing any word will take 2 memory cycles.

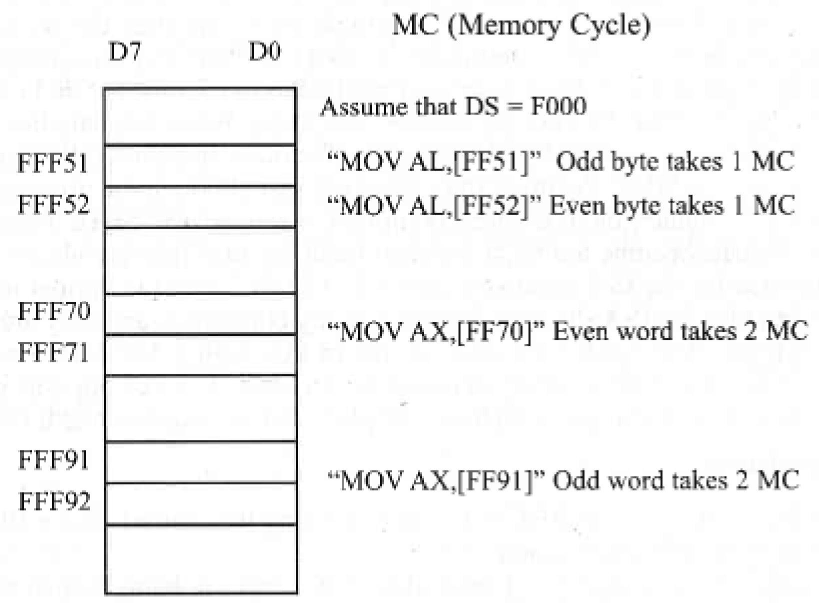

MC (Memory Cycle)
D7
D0
Assume that DS = F000
FFF51
FFF52
"MOV AL,[FF51]" Odd byte takes 1 MC
"MOV AL,[FF52]" Even byte takes 1 MC
FFF70
FFF71
"MOV AX,[FF70]" Even word takes 2 MC
FFF91
FFF92
"MOV AX,[FF91]" Odd word takes 2 MC

Fig: Accessing EVEN & ODD Words in 8-bit CPU

- In the 16- bit system, accessing a word with an even address takes one memory cycle. That is because; one byte is carried on D0-D7 and the other on D8-Dl5 in the same memory cycle.

- But, accessing a word with an odd address requires two memory cycles. For example, see how accessing the word in the instruction "*MOV AX, [F617]*" works as shown in following Fig.

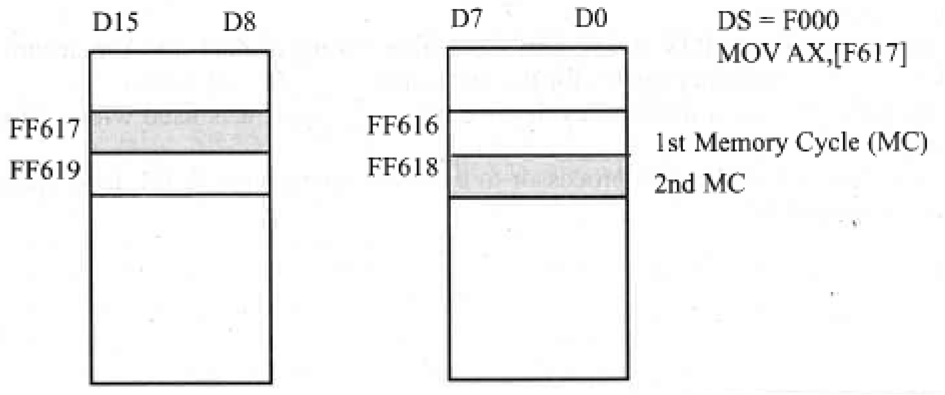


Fig: Accessing an Odd-Addressed Word in 16-bit Processor

- Assuming that DS = F000H in this instruction, the contents of physical memory locations FF6 l7H and FF6l8H are being moved into AX.
- In the first cycle, the 286 CPU accesses location FF617H and puts it in AL.
- In the second cycle, the contents of memory location FF618H are accessed and put into AH.
- Hence, it will be wise to put any words on an even address if the program is going to be run on a 16-bit system.
- A pseudo-instruction is specifically designed for this purpose. It is the EVEN directive and is used as follows:

```
                EVEN
VALUE1          DW      ?
```

- This directive ensures that, the VALUE1, a word-sized operand, is located in an even address location. Hence, an instruction such as "*MOV AX, VALUE1*" will take only a single memory cycle.

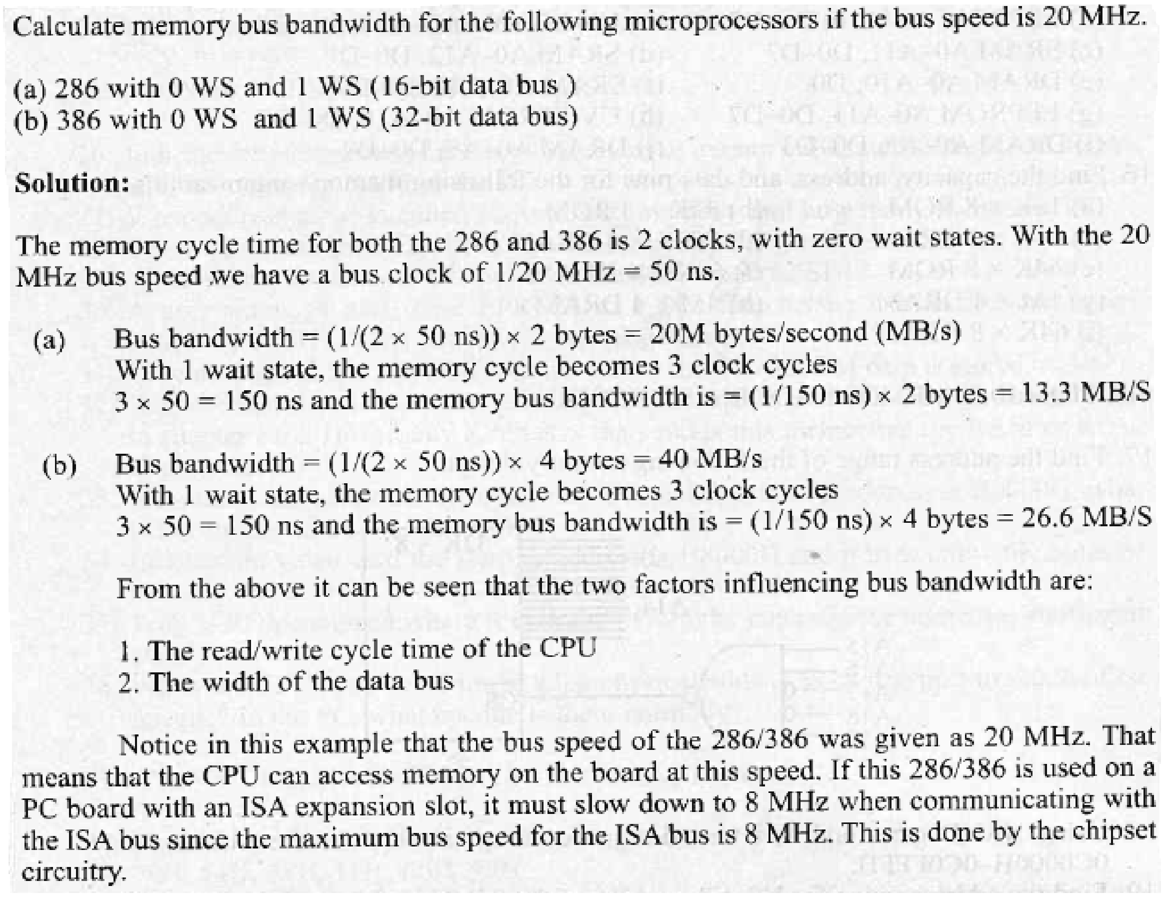
Bus Bandwidth:

» The main advantage of the 16-bit data bus is; doubling of the rate of transfer of information between the CPU and the outside world. The rate of data transfer is generally called *bus bandwidth*. In other words, *bus bandwidth* is a measure of how fast buses transfer information between the CPU and memory or peripherals. The wider the data bus, the higher the bus bandwidth.

» But, the advantage of the wider external data bus comes at the cost of increasing the size of the printed circuit board. Bus bandwidth is measured in MB (megabytes) per second and is calculated as follows:

bus bandwidth = (1/bus cycle time) x bus width in bytes

- In the above formula, bus cycle time can be either memory or I/O cycle time.



Calculate memory bus bandwidth for the following microprocessors if the bus speed is 20 MHz.

(a) 286 with 0 WS and 1 WS (16-bit data bus)
(b) 386 with 0 WS and 1 WS (32-bit data bus)

Solution:

The memory cycle time for both the 286 and 386 is 2 clocks, with zero wait states. With the 20 MHz bus speed we have a bus clock of 1/20 MHz = 50 ns.

(a) Bus bandwidth = (1/(2 × 50 ns)) × 2 bytes = 20M bytes/second (MB/s)
With 1 wait state, the memory cycle becomes 3 clock cycles
3 × 50 = 150 ns and the memory bus bandwidth is = (1/150 ns) × 2 bytes = 13.3 MB/S

(b) Bus bandwidth = (1/(2 × 50 ns)) × 4 bytes = 40 MB/s
With 1 wait state, the memory cycle becomes 3 clock cycles
3 × 50 = 150 ns and the memory bus bandwidth is = (1/150 ns) × 4 bytes = 26.6 MB/S

From the above it can be seen that the two factors influencing bus bandwidth are:

1. The read/write cycle time of the CPU
2. The width of the data bus

Notice in this example that the bus speed of the 286/386 was given as 20 MHz. That means that the CPU can access memory on the board at this speed. If this 286/386 is used on a PC board with an ISA expansion slot, it must slow down to 8 MHz when communicating with the ISA bus since the maximum bus speed for the ISA bus is 8 MHz. This is done by the chipset circuitry.

- There are two ways to increase the bus bandwidth:
    - ✓ Use a wider data bus.
    - ✓ Shorten the bus cycle time.
- While the data bus width has increased from 16-bit in the 80286 to 64-bit in the Pentium, the bus cycle time is reaching a maximum of 133 MHz.

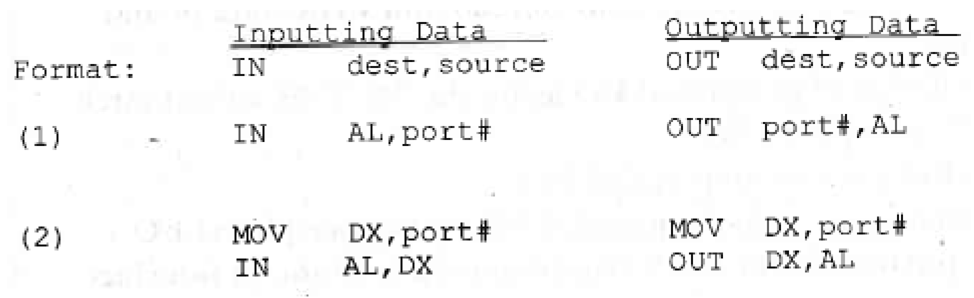
8255 I/O PROGRAMMING

8088 INPUT/OUTPUT INSTRUCTIONS:

- All x86 microprocessors, from the 8088 to the Pentium, can access external devices called ports. This is done using I/O instructions.
- The x86 CPU has I/O space in addition to memory space. While memory can contain Opcode and data, I/O ports contain data only.
- There are two instructions for this purpose: OUT and IN. These instructions can send data from the accumulator (AL or AX) to ports or bring data from ports into the accumulator.
- In accessing ports, we can use an 8-bit or 16-bit data port.

8-bit Data Ports:

- The 8-bit I/O operation of the 8088 is applicable to all x86 CPUs from the 8088 to the Pentium.
- The 8-bit port uses the D0-D7 data bus to communicate with I/O devices.
- In 8-bit port programming, register AL is used as the source of data, when using the OUT instruction; and as the destination, for the IN instruction. This means that to input or output data from any other registers, the data must first be moved to the AL register.
- Instructions OUT and IN have the following formats:



	Inputting Data		Outputting Data	
Format:	IN	dest,source	OUT	dest,source
(1)	IN	AL,port#	OUT	port#,AL
(2)	MOV IN	DX,port# AL,DX	MOV OUT	DX,port# DX,AL

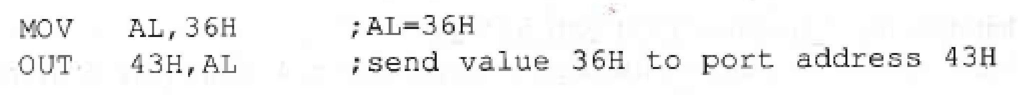
In format (l) –

- ✓ port# is the address of the port and can be from 00 to FFH, allowing up to 256 input and 256 output ports.
- ✓ In this format, the 8-bit port address is carried on address bus A0-A7.
- ✓ No segment register is involved in computing the address.

In format (2) –

- ✓ port# is the address of the port and can be from 0000 to FFFFH, allowing up to 65,536 input and 65,536 output ports.
- ✓ In this format, the 16- bit port address is carried on the address bus A0-A15.
- ✓ The use of a register as a pointer for the port address has an advantage in that the port address can be changed very easily, especially in. cases of dynamic compilations where the port address can be passed to DX.

- » I/O instructions are widely used in programming peripheral devices such as printers, hard disks, and keyboards.
- » The port address can be either 8-bit or 16-bit. For an 8-bit port address, we can use the immediate addressing mode.
- » The following program sends a byte of data to a fixed port address of 43H:



```
MOV   AL,36H        ;AL=36H
OUT   43H,AL        ;send value 36H to port address 43H
```

» The 8-bit address used in immediate addressing mode limits the number of ports to 256 for input plus 256 for output. To have a larger number of ports we must use the 16-bit port address instruction.

» To use the 16-bit port address, *register indirect addressing mode* must be used. The register used for this purpose is DX.

» The following program sends values 55H and AAH to I/O port address 300H (a 16-bit port address).

```
BACK: MOV   DX,300H       ;DX = port address 300H
      MOV   AL,55H
      OUT   DX,AL         ;toggle the bits
      MOV   AL,0AAH
      OUT   DX,AL         ;toggle the bits
      JMP   BACK
```

» We can only use register DX for 16-bit I/O addresses; no other register can be used for this purpose. Also, notice the use of register AL for 8-bit data:

```
MOV DX,378H        ;DX=378 the port address
MOV AL,BL          ;load data into accumulator
OUT DX,AL          ;write contents of AL to port
                   ;whose address is in DX
```

» Just like the OUT instruction, the IN instruction uses the DX register to hold the address and AL to hold the arrived 8-bit data. In other words, DX holds the 16-bit port address while AL receives the 8-bit data brought in from an external port.

» The following program gets data from port address 300H and sends it to port address 302H.

```
MOV   DX,300H      ;load port address
IN    AL,DX        ;bring in data
MOV   DX,302H
OUT   DX,AL        ;send it out
```

In a given 8088-based system, port address 22H is an input port for monitoring the temperature. Write Assembly language instructions to monitor that port continuously for the temperature of 100 degrees. If it reaches 100, then BH should contain 'Y'.

Solution:

```
BACK:    IN    AL,22H   ;get the temperature from port # 22H
         CMP   AL,100        ;is temp = 100?
         JNZ   BACK          ;if not, keep monitoring
         MOV   BH,'Y         ;temp = 100, load 'Y' into BH
```

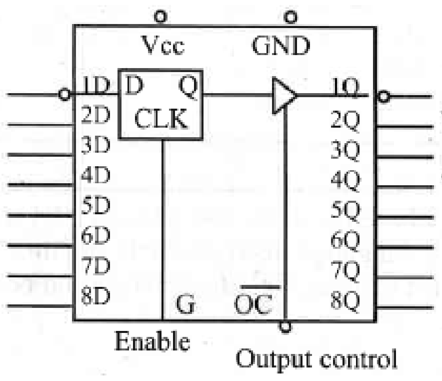
I/O ADDRESS DECODING & DESIGN:

The decoding of I/O ports is done by using TTL logic gates 74LS373 and 74LS244. The following are the steps:

1. The control signals IOR and IOW are used along with the decoders.
2. For an 8-bit port address, A0-A7 is decoded.
3. If the port address is 16-bit (using DX), A0-A15 is decoded.

Using 74LA373 in an Output Port Design:

- In every computer, whenever data is sent out by the CPU via the data bus, the data must be latched by the receiving device. While memories have an internal latch to grab the data, a latching system must be designed for simple I/O ports.
- The 74LS373 can be used for this purpose. Notice in the following Fig. that in order to make the 74LS373 work as a latch, the OC pin must be grounded.



Function Table

Output Control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

Fig: 74LS373 D Latch

- For an output latch, it is common to AND the output of the address decoder with the control signal IOW to provide the latching action as shown in Figure.

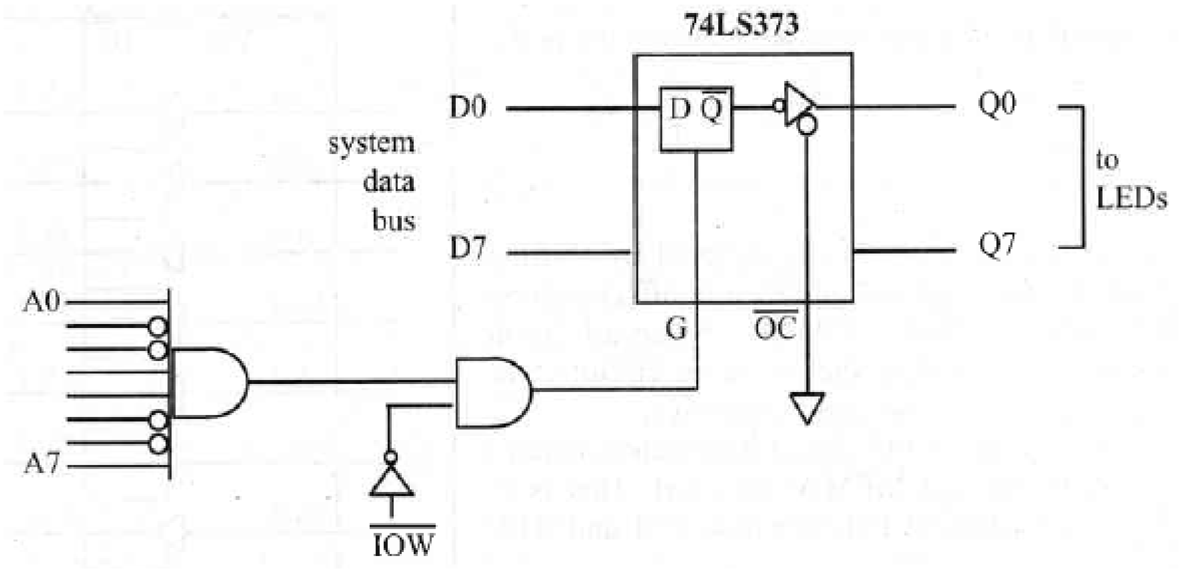


Fig: Design for *"OUT 99H, AL"*

Show the design of an output port with an I/O address of 31FH using the 74LS373.

Solution:

31F9H is decoded, then ANDed with IOW to activate the G pin of the 74LS373 latch. This is shown in Figure below.

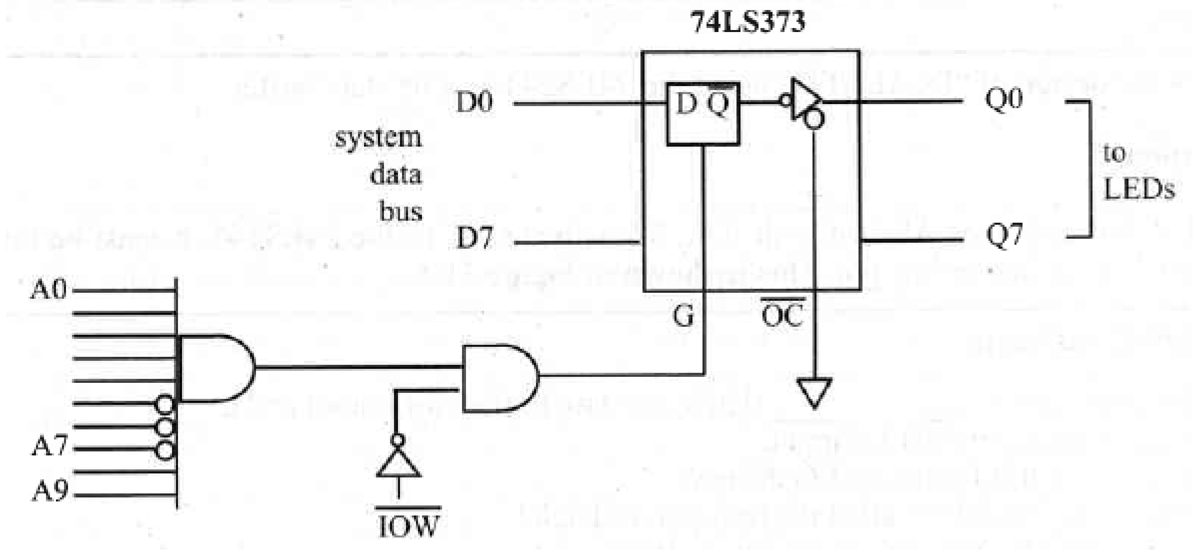


Fig: Design for Output Port Address of 31FH

IN Port Design Using the 74LA244:

- When the data is coming in by way of a data bus, it must come in through a three-state buffer. This is referred to as *tri-stated*. See the following Fig for the internal circuitry of 74LS244.

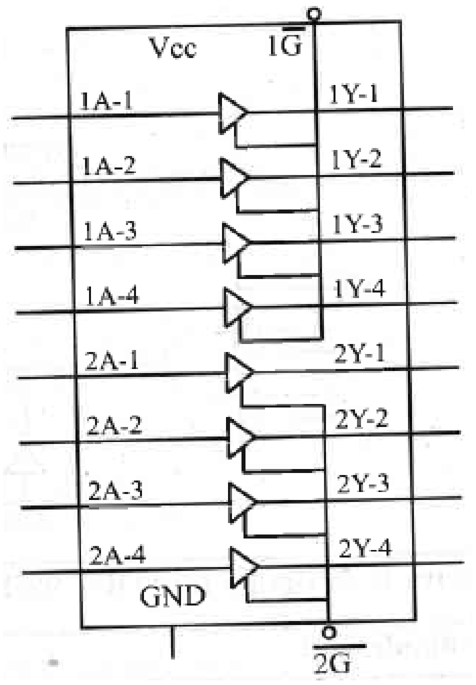


Fig: 74LS244 Octal Buffer

- Here, since 1G and 2G each control only 4 bits of 74LS244, both must be activated for 8 bits input. The following Fig shows the use of 74LS244 as an entry port to the system data bus. In the following Figures, the address decoder and IOR control signal together activate the tri-state input.

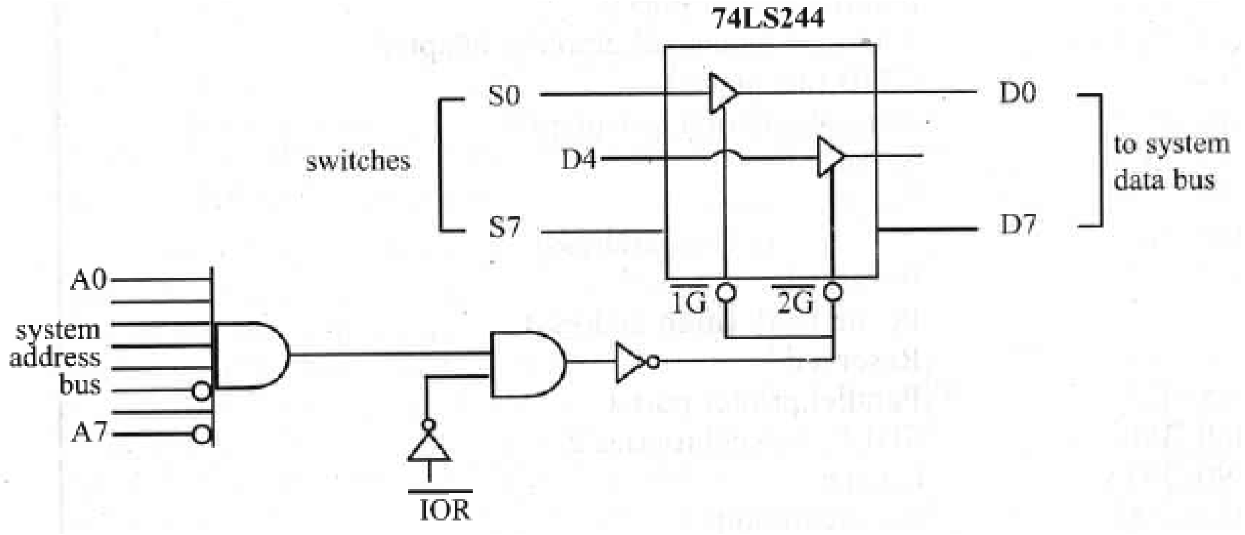


Fig: Input Port Design for *"IN AL, 5FH"*

Show the design of "IN AL,9FH" using the 74LS244 as a tri-state buffer.

Solution:

9FH is decoded, then ANDed with IOR. To activate OC of the 74LS244, it must be inverted since OC is an active-low pin. This is shown in Figure below.

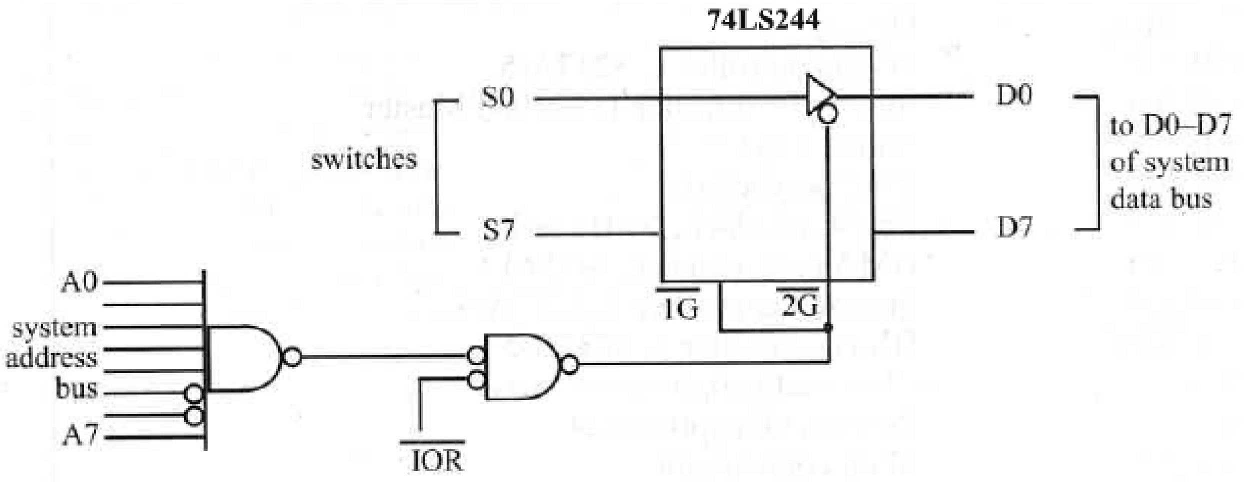


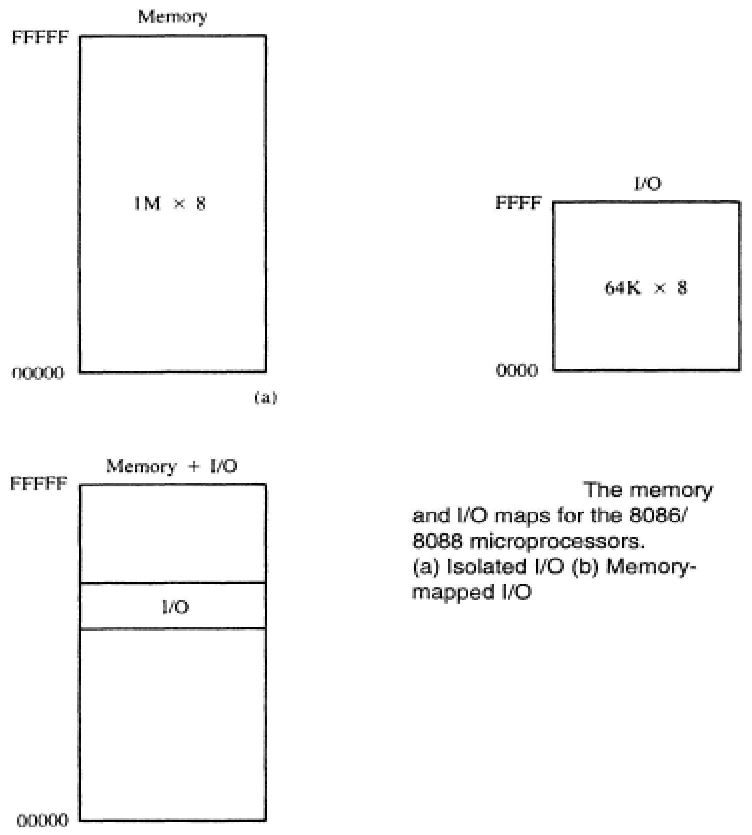
Fig: Design for *"IN AL, 9FH"*

Memory-Mapped I/O:

- Communicating with the I/O devices using IN and OUT instructions is referred to as *peripheral I/O*. Some designers also refer to it as *isolated I/O*.

» Some new RISC processors do not have IN and OUT instructions; they use *memory-mapped I/O*.

» In memory-mapped I/O, a memory location is assigned to be an input and output port.



The memory and I/O maps for the 8086/ 8088 microprocessors. (a) Isolated I/O (b) Memory-mapped I/O

» The following are the differences between peripheral I/O and memory-mapped I/O in x86 PC:

Isolated (Peripheral) I/O	Memory-Mapped I/O
1. The IN and OUT instructions transfer data between the microprocessors accumulator or memory and the I/O device.	1. Instructions that access memory locations are used instead of IN and OUT instructions: *MOV AL, [2000]* will access the input port & *MOV [2000], AL* will access the output port.

2. Only A0-A15 are decoded; Hence, DS initialization is not required; decoding circuitry may be less expensive.	2. Entire 20-bit address, A0-A19, must be decoded (decoding circuitry is expensive); Hence DS must be loaded before accessing memory-mapped I/O:
	`MOV AX,3000H ;load the segment value`
	`MOV DS,AX`
	`MOV AL,[5000] ;get a byte from loc. 35000H`

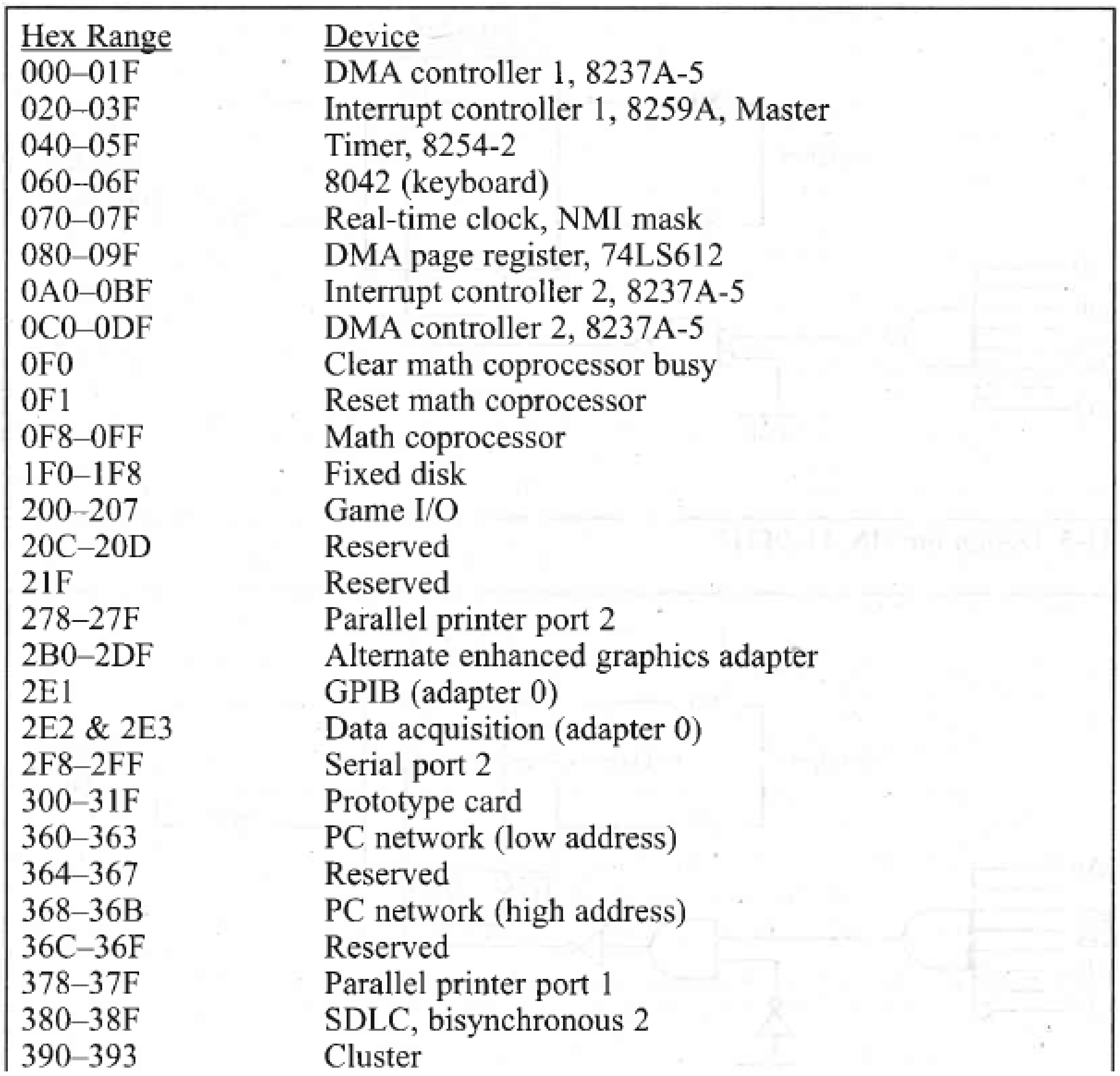
3. IOR and IOW control signals are used.	3. MEMR and MEMW control signals are used.
4. Limited only to 65,536 input ports	4. The number of ports can be as high as 2^{20} (1,048,576).

and 65,536 output ports.	
5. Data should be moved to accumulator for any kind of operations.	5. Arithmetic and logic operations can be performed directly, without moving data to accumulator.
6. The user can expand the memory to its full size without using any memory space for I/O devices.	6. Uses memory address space, which could lead to memory space fragmentation.

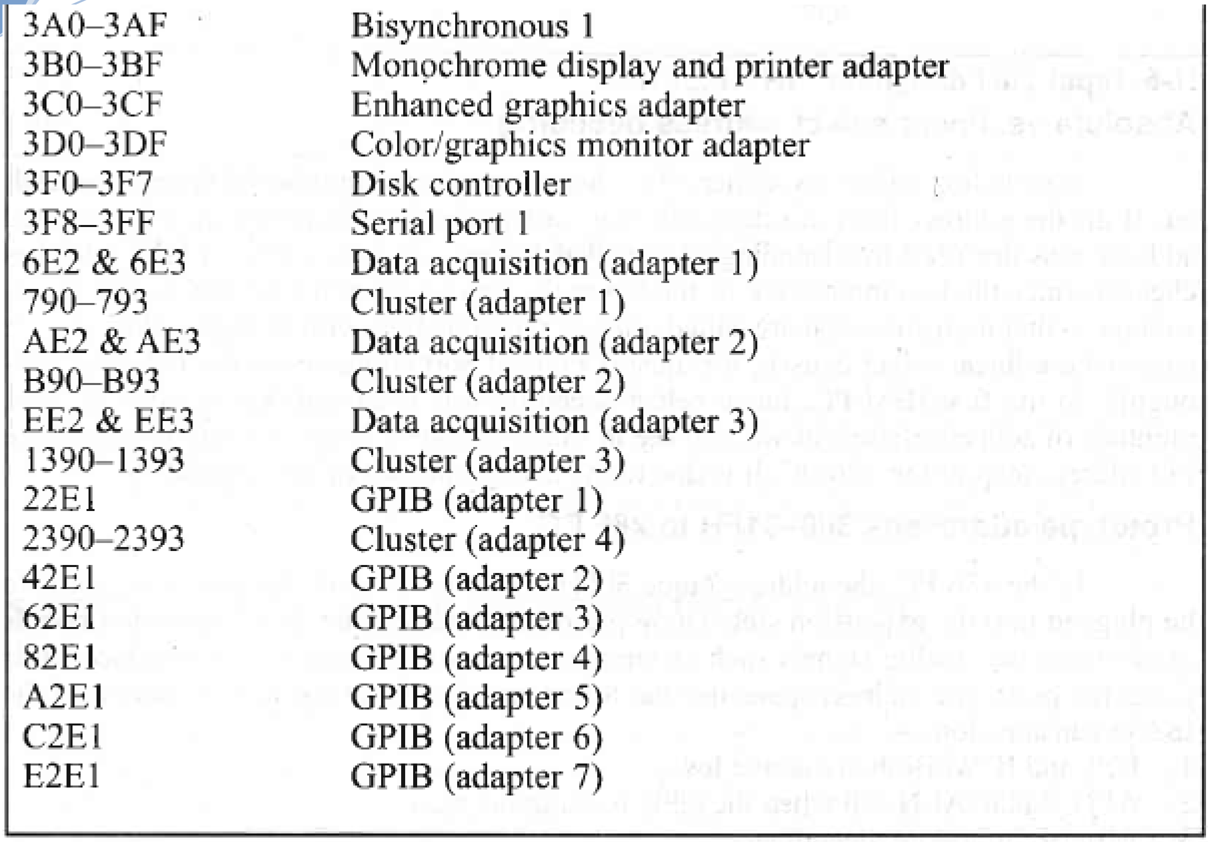
I/O ADDRESS MAP OF x86 PCs:

Any system that needs to be compatible with the x86 IBM PC must follow the I/O map of the following Table:

Table: I/O Map for x86 PC



Hex Range	Device
000–01F	DMA controller 1, 8237A-5
020–03F	Interrupt controller 1, 8259A, Master
040–05F	Timer, 8254-2
060–06F	8042 (keyboard)
070–07F	Real-time clock, NMI mask
080–09F	DMA page register, 74LS612
0A0–0BF	Interrupt controller 2, 8237A-5
0C0–0DF	DMA controller 2, 8237A-5
0F0	Clear math coprocessor busy
0F1	Reset math coprocessor
0F8–0FF	Math coprocessor
1F0–1F8	Fixed disk
200–207	Game I/O
20C–20D	Reserved
21F	Reserved
278–27F	Parallel printer port 2
2B0–2DF	Alternate enhanced graphics adapter
2E1	GPIB (adapter 0)
2E2 & 2E3	Data acquisition (adapter 0)
2F8–2FF	Serial port 2
300–31F	Prototype card
360–363	PC network (low address)
364–367	Reserved
368–36B	PC network (high address)
36C–36F	Reserved
378–37F	Parallel printer port 1
380–38F	SDLC, bisynchronous 2
390–393	Cluster



3A0–3AF	Bisynchronous 1
3B0–3BF	Monochrome display and printer adapter
3C0–3CF	Enhanced graphics adapter
3D0–3DF	Color/graphics monitor adapter
3F0–3F7	Disk controller
3F8–3FF	Serial port 1
6E2 & 6E3	Data acquisition (adapter 1)
790–793	Cluster (adapter 1)
AE2 & AE3	Data acquisition (adapter 2)
B90–B93	Cluster (adapter 2)
EE2 & EE3	Data acquisition (adapter 3)
1390–1393	Cluster (adapter 3)
22E1	GPIB (adapter 1)
2390–2393	Cluster (adapter 4)
42E1	GPIB (adapter 2)
62E1	GPIB (adapter 3)
82E1	GPIB (adapter 4)
A2E1	GPIB (adapter 5)
C2E1	GPIB (adapter 6)
E2E1	GPIB (adapter 7)

Absolute vs. Linear Select Address Decoding:

- In decoding addresses, either all the address lines or a selected number of them are decoded.
    - If all the address lines are decoded, it is called *absolute decoding*.
    - If only selected address pins are used for decoding, it is called *linear select decoding* – This is cheaper due to the less number of input and the fewer the gates needed for decoding. The disadvantage is that it creates what are called *aliases*, the same port with multiple addresses. Hence, port address documentation is necessary.

Portable Addresses 300 – 31FH in x86 PC:

In the x86 PC, the address range 300H – 31FH is set aside for prototype cards to be plugged into the expansion slot. These prototype cards can be data acquisition boards used to monitor analog signals such as temperature, pressure, and so on. Interface cards using the prototype address space use the following signals on the 62-pin section of the ISA expansion slot:

1. IOR and IOW. Both are active low.
2. AEN signal: AEN = 0 when the CPU is using the bus.
3. A0-A9 for address decoding.

Use of Simple Logic Gates as Address Decoders:

The following Fig shows the circuit design for a 74LS373 latch connected to port address 300H of an x86 PC via an ISA expansion slot. Notice the use of signals A0-A9 and AEN. AEN is low when the x86 microprocessor is in control of the buses. Here, we are using simple logic gates such as NAND and inverter gates for the I/O address decoder. These can be replaced with the 74LS138 chip because the 74LS138 is a group of NAND gates in a single chip.

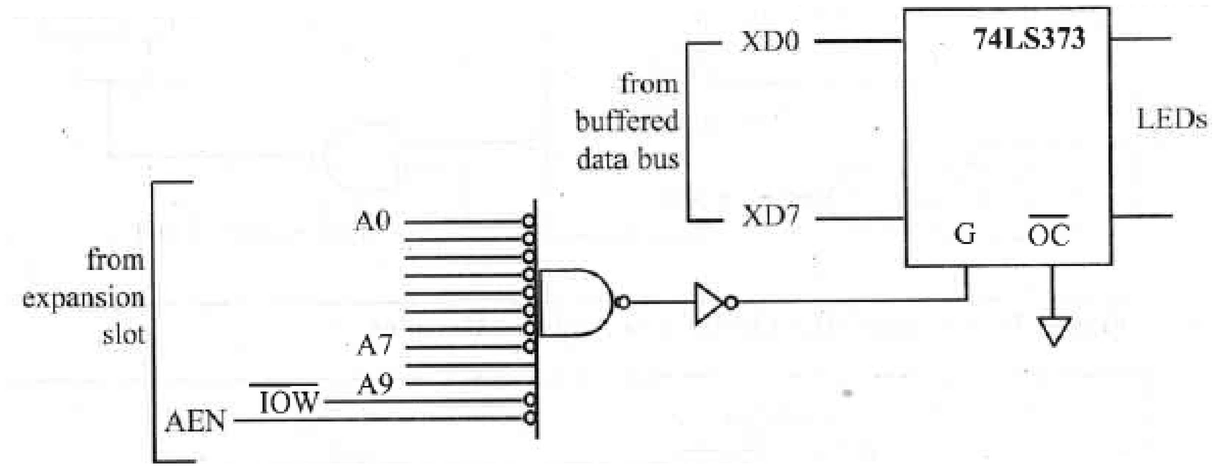
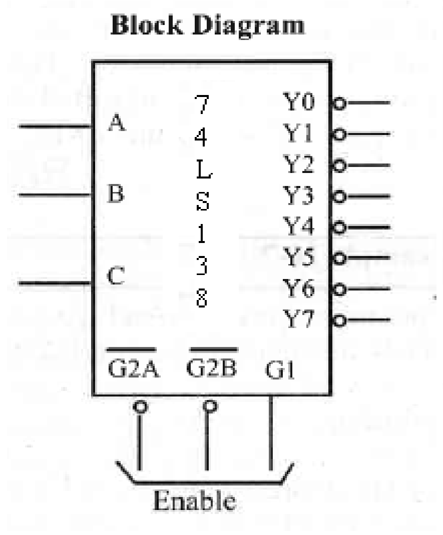
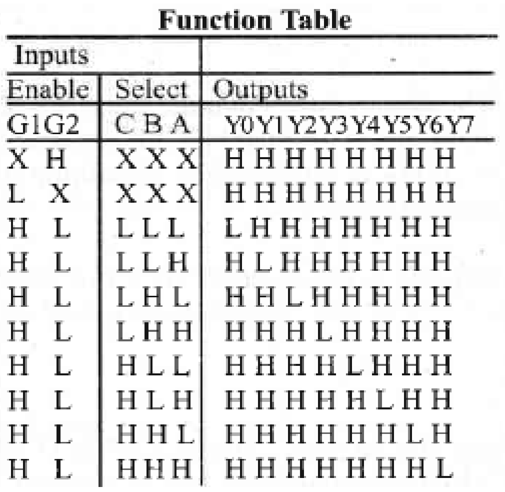


Fig: Using Simple Logic Gates for I/O Address Decoder (I/O Address 300H)

Use of 74LS138 as Decoder:

The following Fig shows the 74LS138.





Function Table

Inputs			
Enable		Select	Outputs
G1	G2	C B A	Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7
X	H	X X X	H H H H H H H H
L	X	X X X	H H H H H H H H
H	L	L L L	L H H H H H H H
H	L	L L H	H L H H H H H H
H	L	L H L	H H L H H H H H
H	L	L H H	H H H L H H H H
H	L	H L L	H H H H L H H H
H	L	H L H	H H H H H L H H
H	L	H H L	H H H H H H L H
H	L	H H H	H H H H H H H L

The following Fig is an example of the use of a 74LA138 for an I/O address decoder.

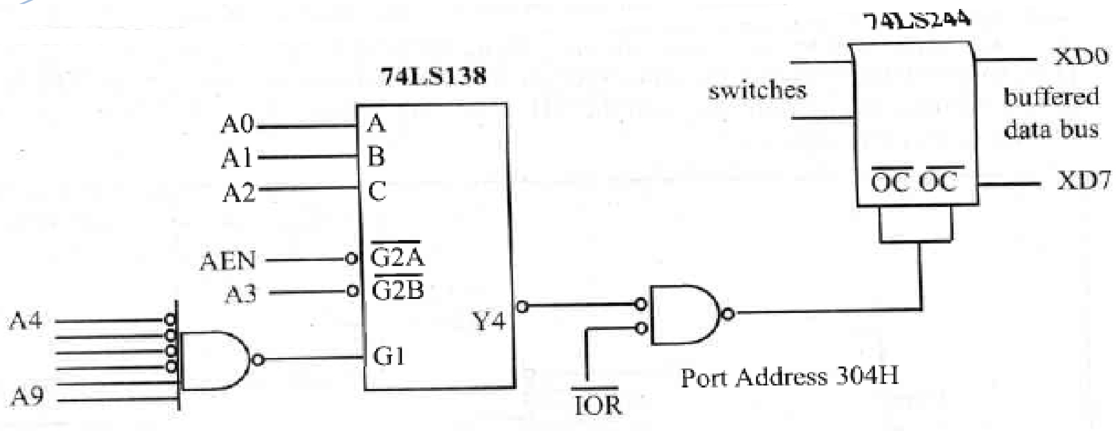


Fig: Using 74LS138 for I/O Address Decoding

- ✓ This is an address decoding for an input port located at address 304H.
- ✓ The Y4 output, together with the IOR signal, controls the 74LS244 input buffer.
- ✓ Note that, each Y output can control a single I/O device.

IBM PC I/O Address Decoder:

The following Fig shows a 74LS138 chip used as an I/O address decoder in the original IBM PC.

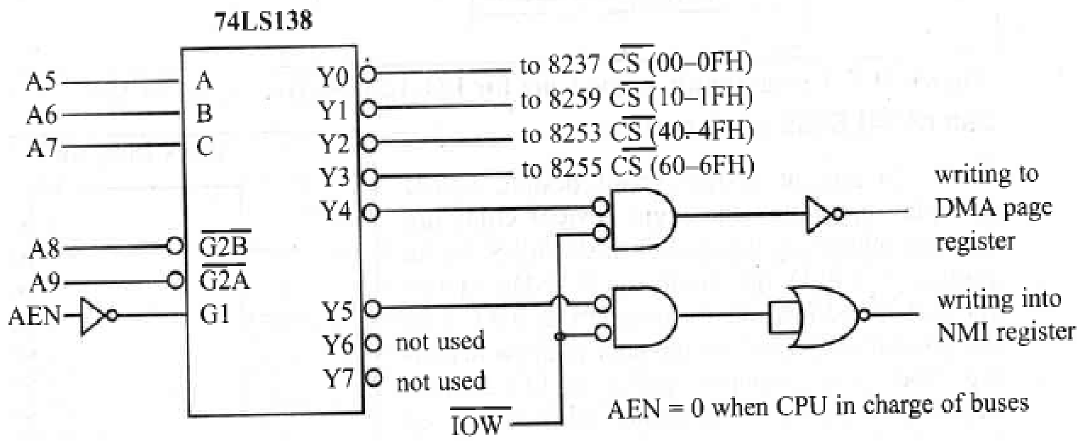
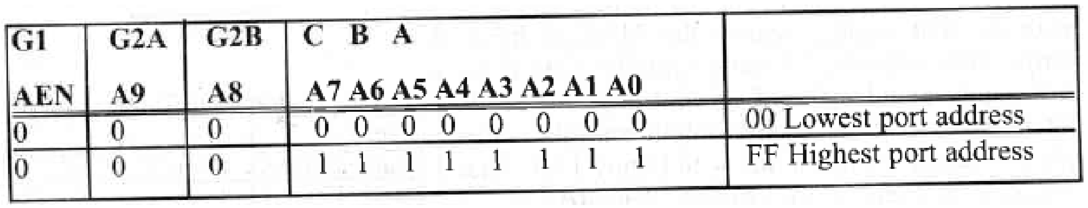


Fig: Port Address Decoding in the Original IBM PC

- ✓ Notice that, while A0 to A4 go to individual peripheral input addresses, A5, A6, and A7 are responsible for the selection of outputs Y0 to Y7.
- ✓ In order to enable the 74LS138, pins A8, A9, and AEN all must be low. While A8 and A9 will directly affect the port address calculations, AEN is low only when the x86 is in control of the system bus (see the following Table).

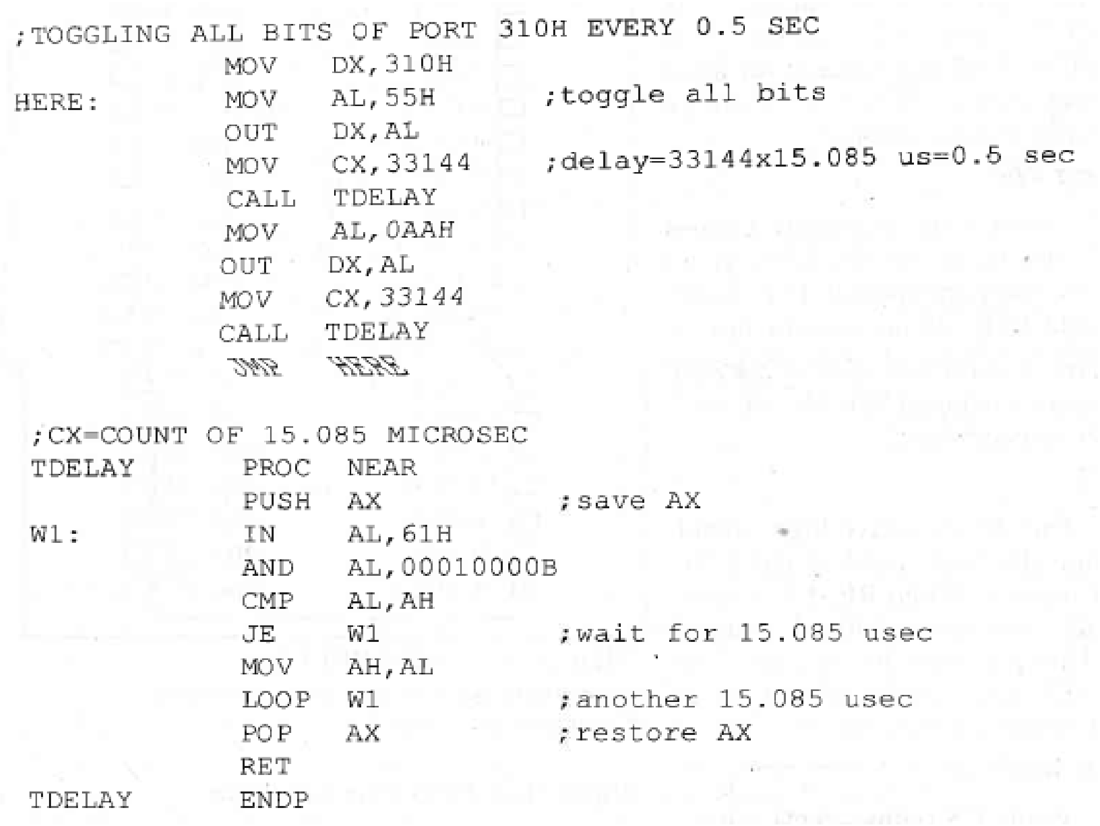
Table: Port Address Decoding Table on the Original PC



G1	G2A	G2B	C B A	
AEN	A9	A8	A7 A6 A5 A4 A3 A2 A1 A0	
0	0	0	0 0 0 0 0 0 0 0	00 Lowest port address
0	0	0	1 1 1 1 1 1 1 1	FF Highest port address

Port 61H and Time Delay Generation:

- In order to maintain compatibility with the IBM PC and run operating systems such as MS-DOS and Windows, the assignment of I/O port addresses must follow the standard.
- Port 61H is a widely used port. We can use this port to generate a time delay which will work in any PC with any type of processor from the 286 to the Pentium.
- I/O port 61H has eight bits (D0-D7). Bit D4 is of particular interest to us. In all 286 and higher PCs bit D4 of port 61H changes its state every 15.085 microseconds (µs) (stays low for 15.085 µs and then changes to high and stay high for the same amount of time before it goes low again).
- This toggling of bit D4 goes on indefinitely as long as the PC is on.
- The following program shows how to use port 61H to generate a delay of 1/2 second. In this program all the bits of port 310H are toggled with a 1/2 second delay in between.



```
;TOGGLING ALL BITS OF PORT 310H EVERY 0.5 SEC
              MOV    DX,310H
HERE:         MOV    AL,55H         ;toggle all bits
              OUT    DX,AL
              MOV    CX,33144       ;delay=33144x15.085 us=0.5 sec
              CALL   TDELAY
              MOV    AL,0AAH
              OUT    DX,AL
              MOV    CX,33144
              CALL   TDELAY
              JMP    HERE

;CX=COUNT OF 15.085 MICROSEC
TDELAY        PROC   NEAR
              PUSH   AX                ;save AX
W1:           IN     AL,61H
              AND    AL,00010000B
              CMP    AL,AH
              JE     W1                ;wait for 15.085 usec
              MOV    AH,AL
              LOOP   W1                ;another 15.085 usec
              POP    AX                ;restore AX
              RET
TDELAY        ENDP
```

Notice that, when port 61H is read, all the bits are masked except D4. The program waits for D4 to change every 15.085 µs before it loops again.

PROGRAMMING & INTERFACING THE 8255:

The 8255 is –

» a widely used 40-pin DIP I/O chip.

» Having three separately accessible ports, A, B, and C, which can be programmed to be input or output port, hence the name PPI (*programmable peripheral interface*).

- They can also be changed dynamically, in contrast to the 74LS244 and 74LS373, which are hard-wired.

Port A (PA0-PA7):

- This 8-bit port A can be programmed all as input or all as output.

Port B (PB0-PB7):

- This 8-bit port B can be programmed all as input or all as output.

Port C (PC0-PC7):

- This 8-bit port C can be programmed all as input or all as output.
- It can also be split into two parts; CU (upper bits PC4-PC7) and CL (lower bits PC0-PC3). Each can be used as input or output.
- Any bit of Port C can be programmed individually.

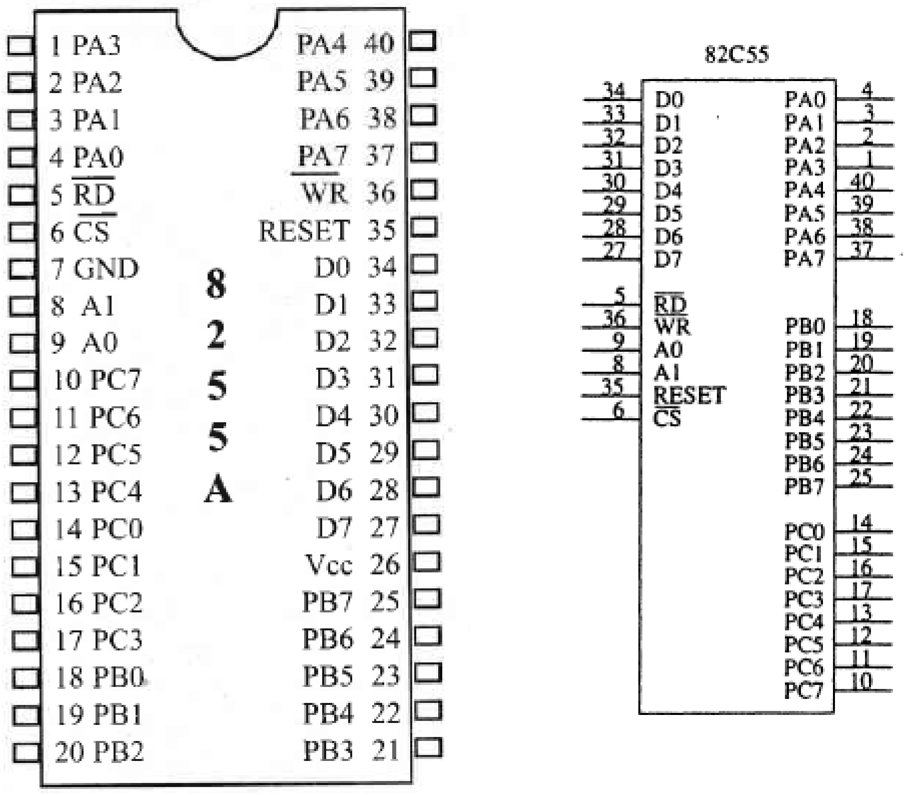


Fig: 8255 PPI Chip

RD and WR:

- Active low input signals to 8255.
- If 8255 is using peripheral I/O design, IOR and IOW of the system bus are connected to these two pins.
- If 8255 is using memory-mapped I/O, MEMR and MEMW of the system bus will activate these two pins.

RESET:

» Active high signal input to 8255.

» Used to clear the control register.

» When RESET is activated, all the ports are initialized as input ports.

» This pin must be connected to the RESET output of the system bus, or grounded, making it inactive.

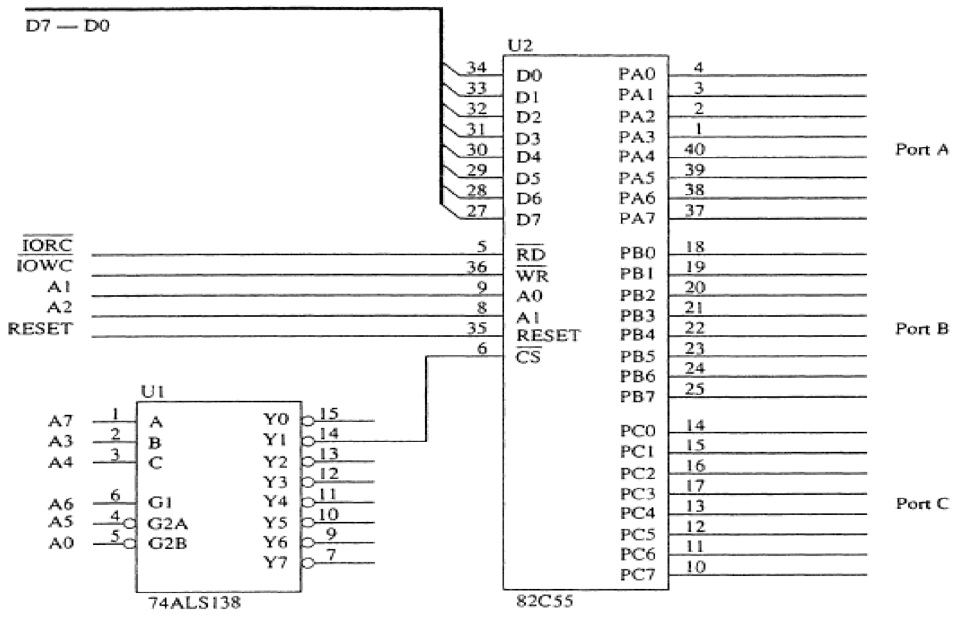
CS	A1	A0	Selects
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	x	x	8255 is not selected

A0, A1, and CS:

» CS (chip select) selects the entire chip.

» Address pins A0 and A1 selects specific port within the 8255.

» These three pins are used to access ports A, B, C, or the control register; as shown in the table:



Mode Selection of the 8255A:

The ports (A, B, and C) of the 8255 can be programmed in various modes, as shown in the following Fig.

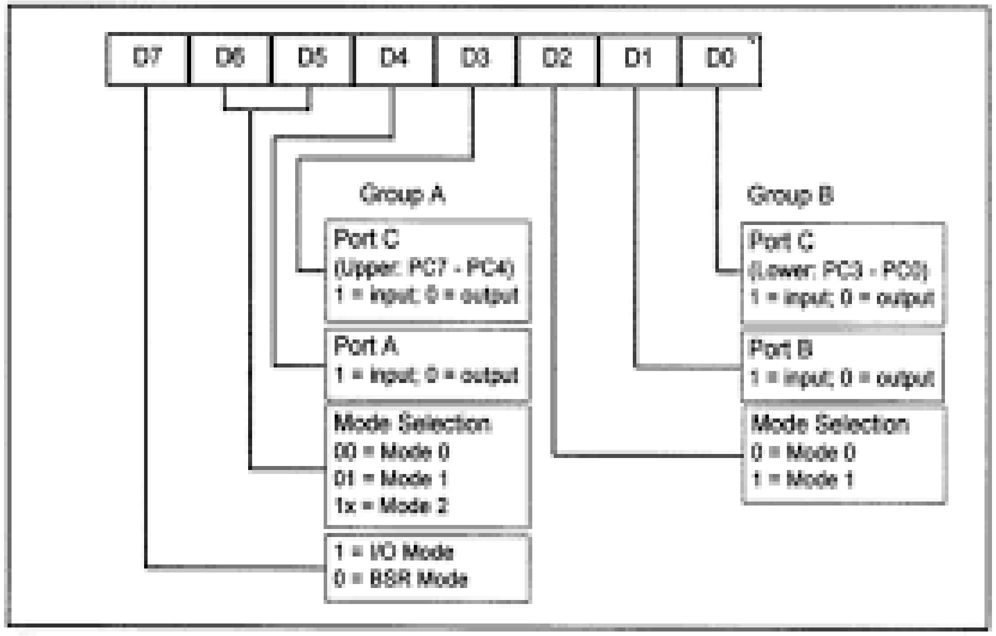
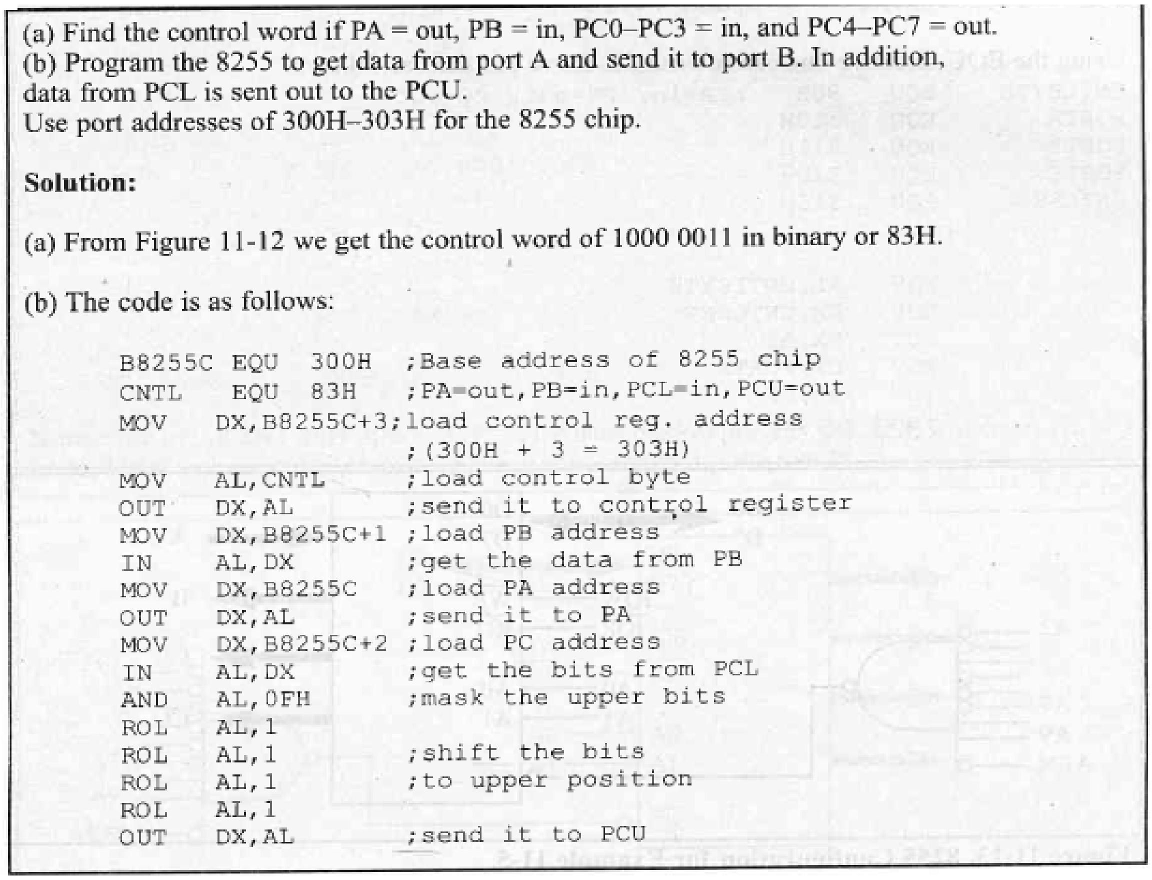


Fig: Control Word Format

Mode 0, the simple I/O mode, is the most widely used mode. In this mode, any of the ports A, B, CU, and CL can be programmed as input or output. In this mode, all bits are out or all are in. In other words, there is no control of individual bits.



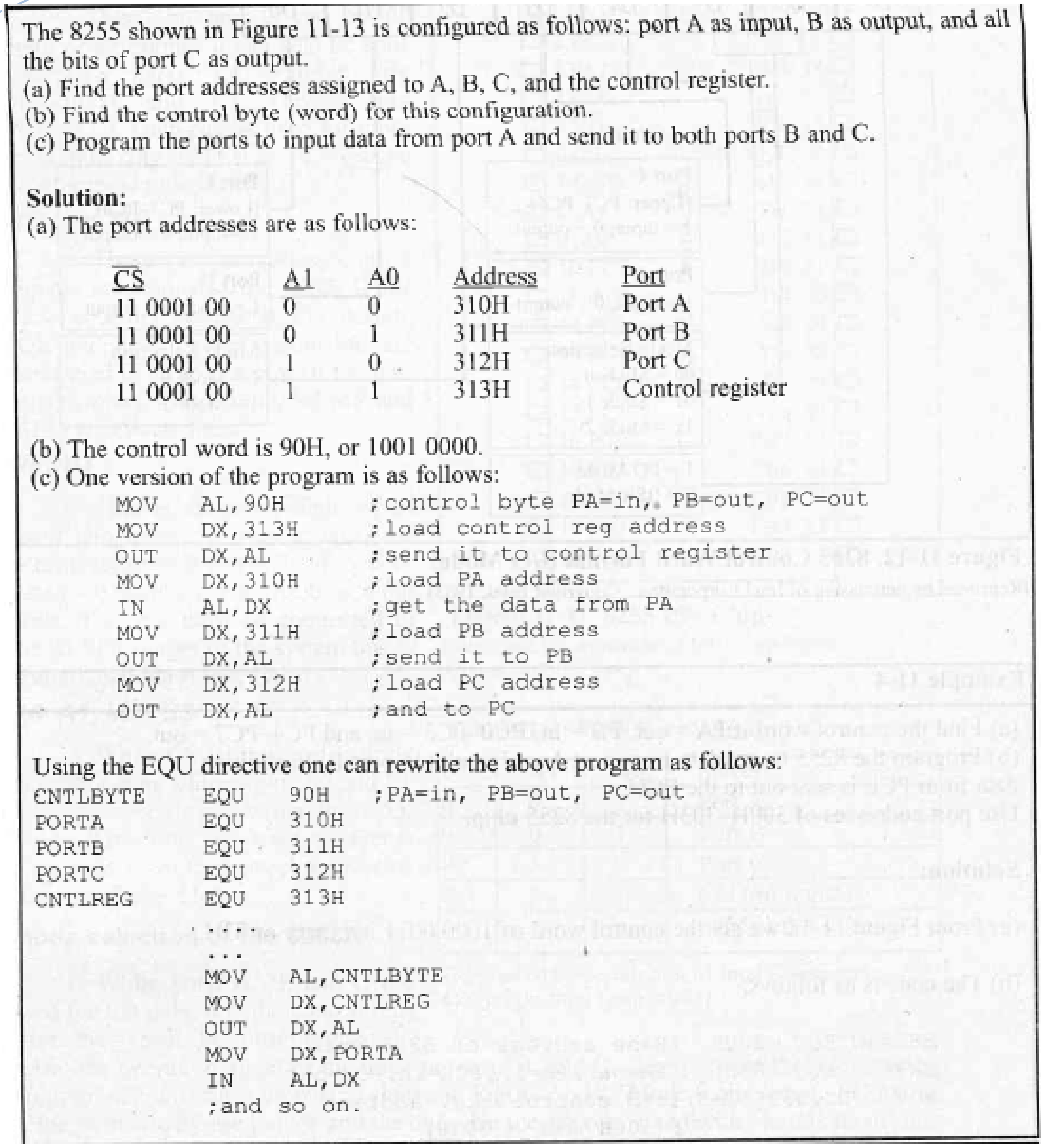
(a) Find the control word if PA = out, PB = in, PC0–PC3 = in, and PC4–PC7 = out.
(b) Program the 8255 to get data from port A and send it to port B. In addition, data from PCL is sent out to the PCU.
Use port addresses of 300H–303H for the 8255 chip.

Solution:

(a) From Figure 11-12 we get the control word of 1000 0011 in binary or 83H.

(b) The code is as follows:

```
B8255C EQU  300H  ;Base address of 8255 chip
CNTL    EQU  83H   ;PA=out,PB=in,PCL=in,PCU=out
MOV    DX,B8255C+3;load control reg. address
                  ;(300H + 3 = 303H)
MOV    AL,CNTL     ;load control byte
OUT    DX,AL       ;send it to control register
MOV    DX,B8255C+1 ;load PB address
IN     AL,DX       ;get the data from PB
MOV    DX,B8255C   ;load PA address
OUT    DX,AL       ;send it to PA
MOV    DX,B8255C+2 ;load PC address
IN     AL,DX       ;get the bits from PCL
AND    AL,0FH      ;mask the upper bits
ROL    AL,1
ROL    AL,1        ;shift the bits
ROL    AL,1        ;to upper position
ROL    AL,1
OUT    DX,AL       ;send it to PCU
```



The 8255 shown in Figure 11-13 is configured as follows: port A as input, B as output, and all the bits of port C as output.
(a) Find the port addresses assigned to A, B, C, and the control register.
(b) Find the control byte (word) for this configuration.
(c) Program the ports to input data from port A and send it to both ports B and C.

Solution:
(a) The port addresses are as follows:

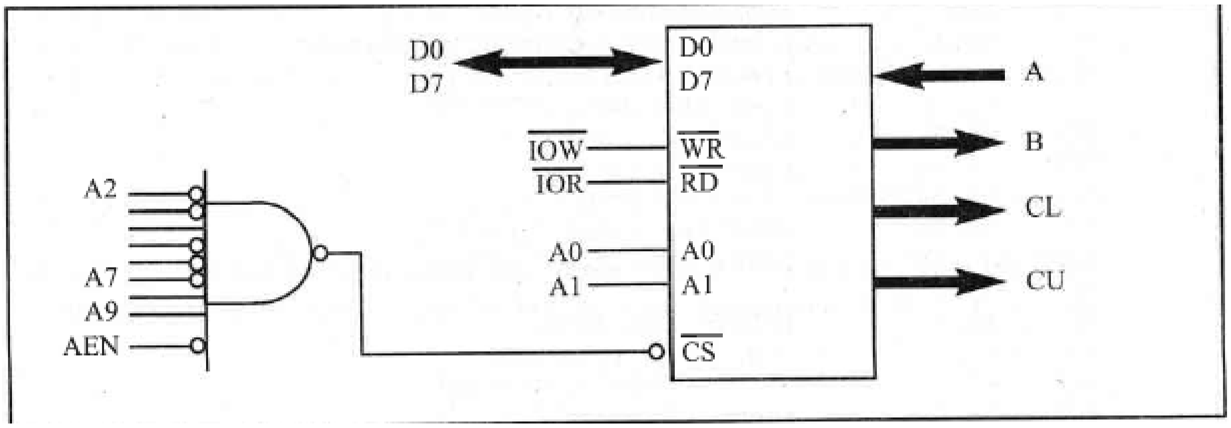
$\overline{CS}$	A1	A0	Address	Port
11 0001 00	0	0	310H	Port A
11 0001 00	0	1	311H	Port B
11 0001 00	1	0	312H	Port C
11 0001 00	1	1	313H	Control register

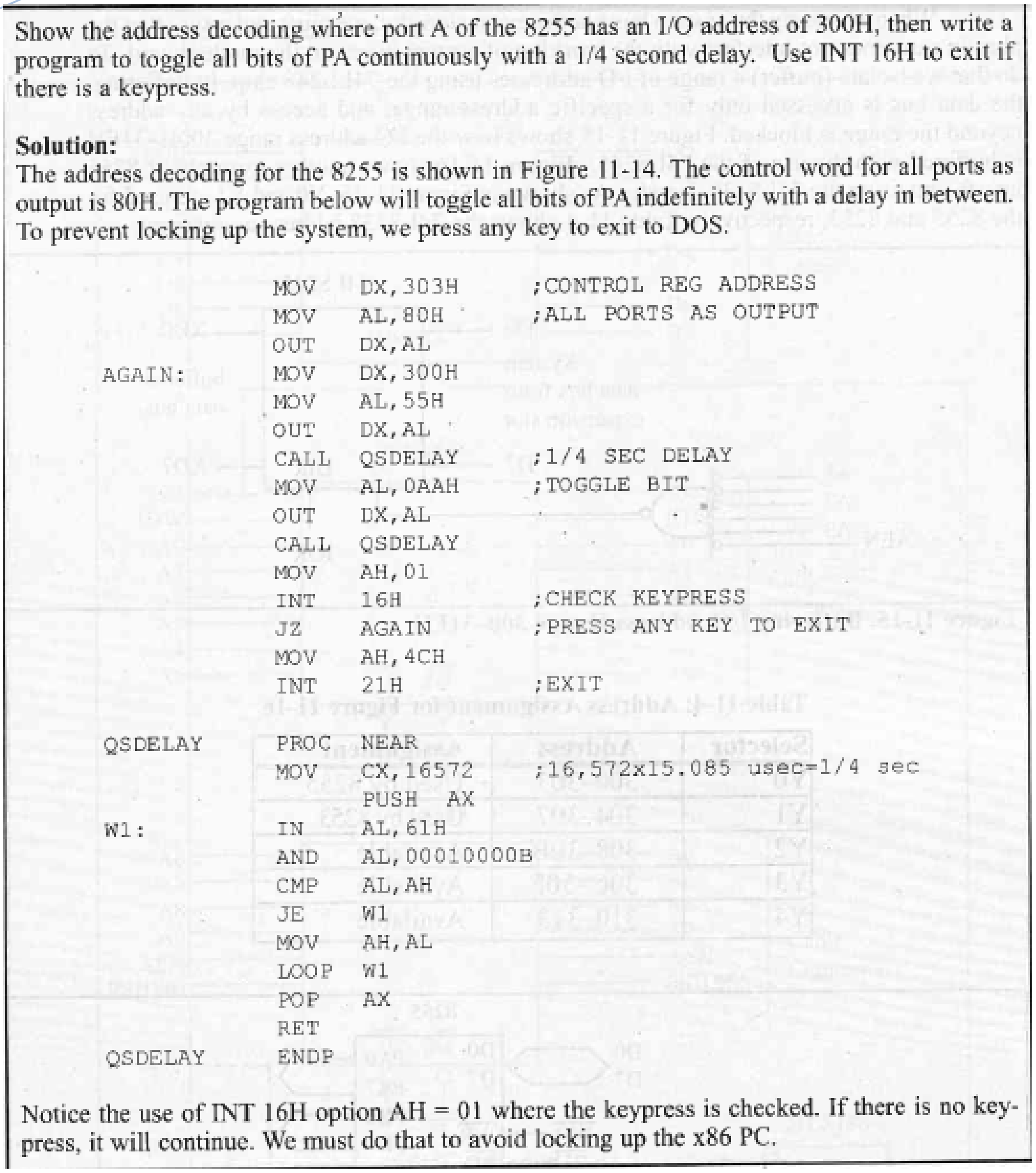
(b) The control word is 90H, or 1001 0000.
(c) One version of the program is as follows:

```
        MOV    AL,90H      ;control byte PA=in, PB=out, PC=out
        MOV    DX,313H     ;load control reg address
        OUT    DX,AL       ;send it to control register
        MOV    DX,310H     ;load PA address
        IN     AL,DX       ;get the data from PA
        MOV    DX,311H     ;load PB address
        OUT    DX,AL       ;send it to PB
        MOV    DX,312H     ;load PC address
        OUT    DX,AL       ;and to PC
```

Using the EQU directive one can rewrite the above program as follows:

```
CNTLBYTE    EQU    90H    ;PA=in, PB=out, PC=out
PORTA       EQU    310H
PORTB       EQU    311H
PORTC       EQU    312H
CNTLREG     EQU    313H
            ...
            MOV    AL,CNTLBYTE
            MOV    DX,CNTLREG
            OUT    DX,AL
            MOV    DX,PORTA
            IN     AL,DX
            ;and so on.
```



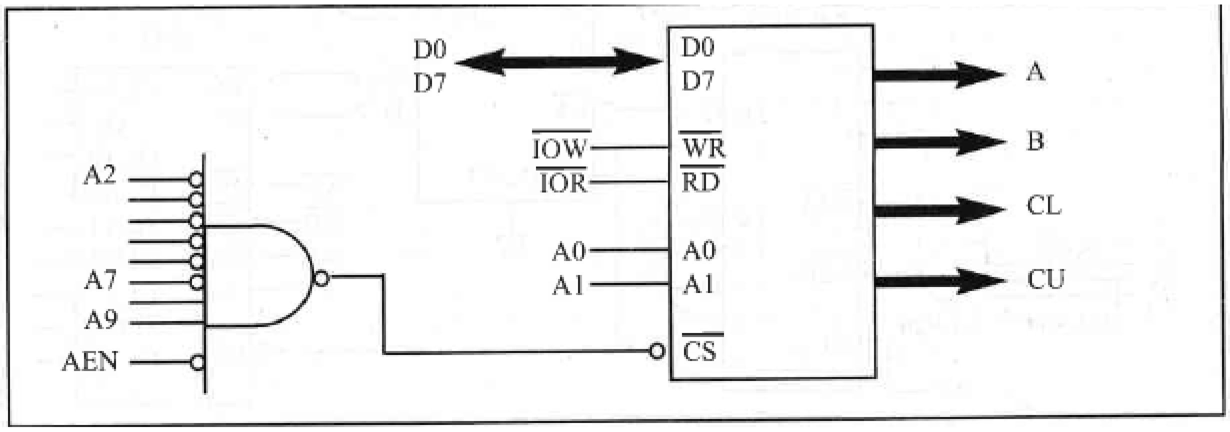


Show the address decoding where port A of the 8255 has an I/O address of 300H, then write a program to toggle all bits of PA continuously with a 1/4 second delay. Use INT 16H to exit if there is a keypress.

Solution:
The address decoding for the 8255 is shown in Figure 11-14. The control word for all ports as output is 80H. The program below will toggle all bits of PA indefinitely with a delay in between. To prevent locking up the system, we press any key to exit to DOS.

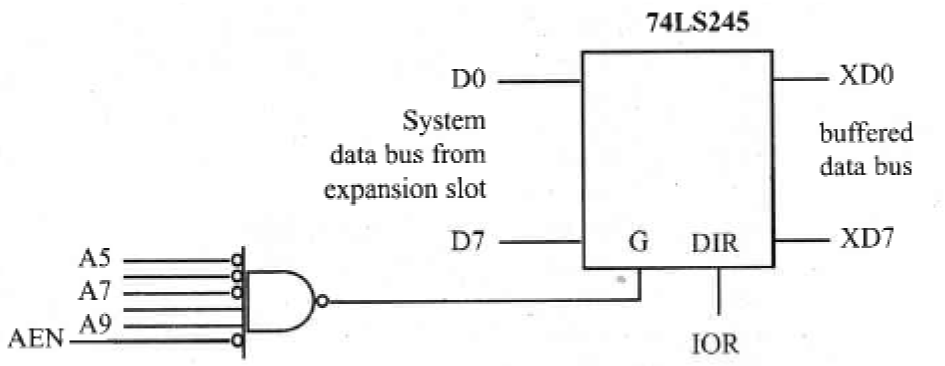
```
            MOV     DX,303H         ;CONTROL REG ADDRESS
            MOV     AL,80H          ;ALL PORTS AS OUTPUT
            OUT     DX,AL
AGAIN:      MOV     DX,300H
            MOV     AL,55H
            OUT     DX,AL
            CALL    QSDELAY         ;1/4 SEC DELAY
            MOV     AL,0AAH         ;TOGGLE BIT
            OUT     DX,AL
            CALL    QSDELAY
            MOV     AH,01
            INT     16H             ;CHECK KEYPRESS
            JZ      AGAIN           ;PRESS ANY KEY TO EXIT
            MOV     AH,4CH
            INT     21H             ;EXIT

QSDELAY     PROC    NEAR
            MOV     CX,16572        ;16,572x15.085 usec=1/4 sec
                    PUSH    AX
W1:         IN      AL,61H
            AND     AL,00010000B
            CMP     AL,AH
            JE      W1
            MOV     AH,AL
            LOOP    W1
            POP     AX
            RET
QSDELAY     ENDP
```

Notice the use of INT 16H option AH = 01 where the keypress is checked. If there is no keypress, it will continue. We must do that to avoid locking up the x86 PC.



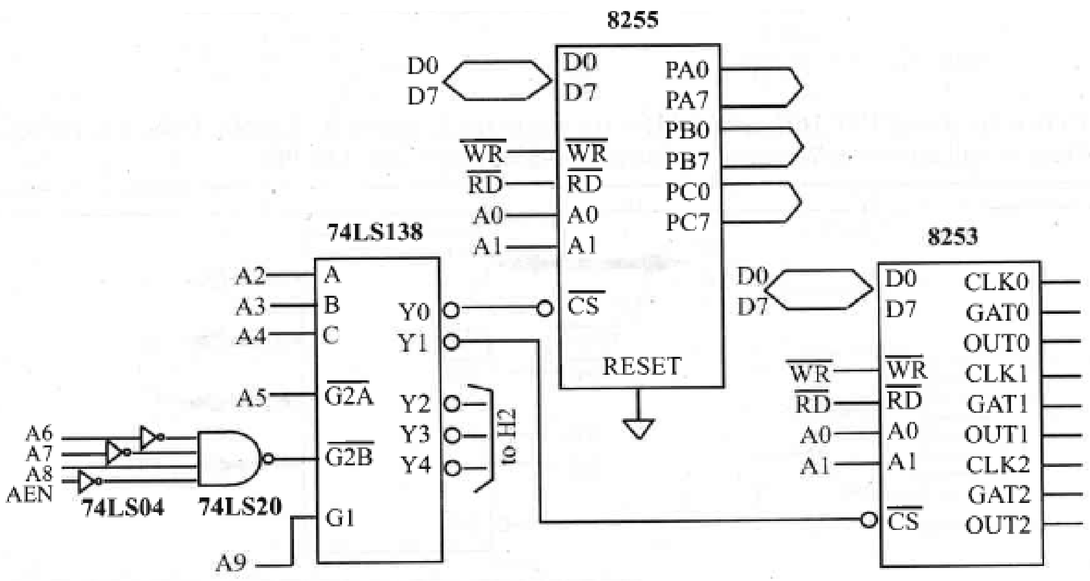
Buffering 300 – 31FH Address Range:

- When accessing the system bus via the expansion slot; we must make sure that the plug-in card does not interfere with the working of system buses on the motherboard.
- To do that we isolate (buffer) a range of I/O addresses using the 74LS245 chip.
- In buffering, the data bus is accessed only for a specific address range, and access by any address beyond the range is blocked.
- The following Fig shows how the I/O address range 300H-31FH is buffered with the use of the 74LS245.



- The following Fig shows another example of 8255 interfacing using the 74LS138 decoder. As shown in the Fig., Y0 and Y1 are used for the 8255 and 8253, respectively. The Table shows the 74LS 138 address assignment.

Selector	Address	Assignment
Y0	300–303	Used by 8255
Y1	304–307	Used by 8253
Y2	308–30B	Available
Y3	30C–30F	Available
Y4	310–313	Available



- The following Fig shows the circuit for buffering all the buses. The 74LS244 is used to boost the address and control signals.

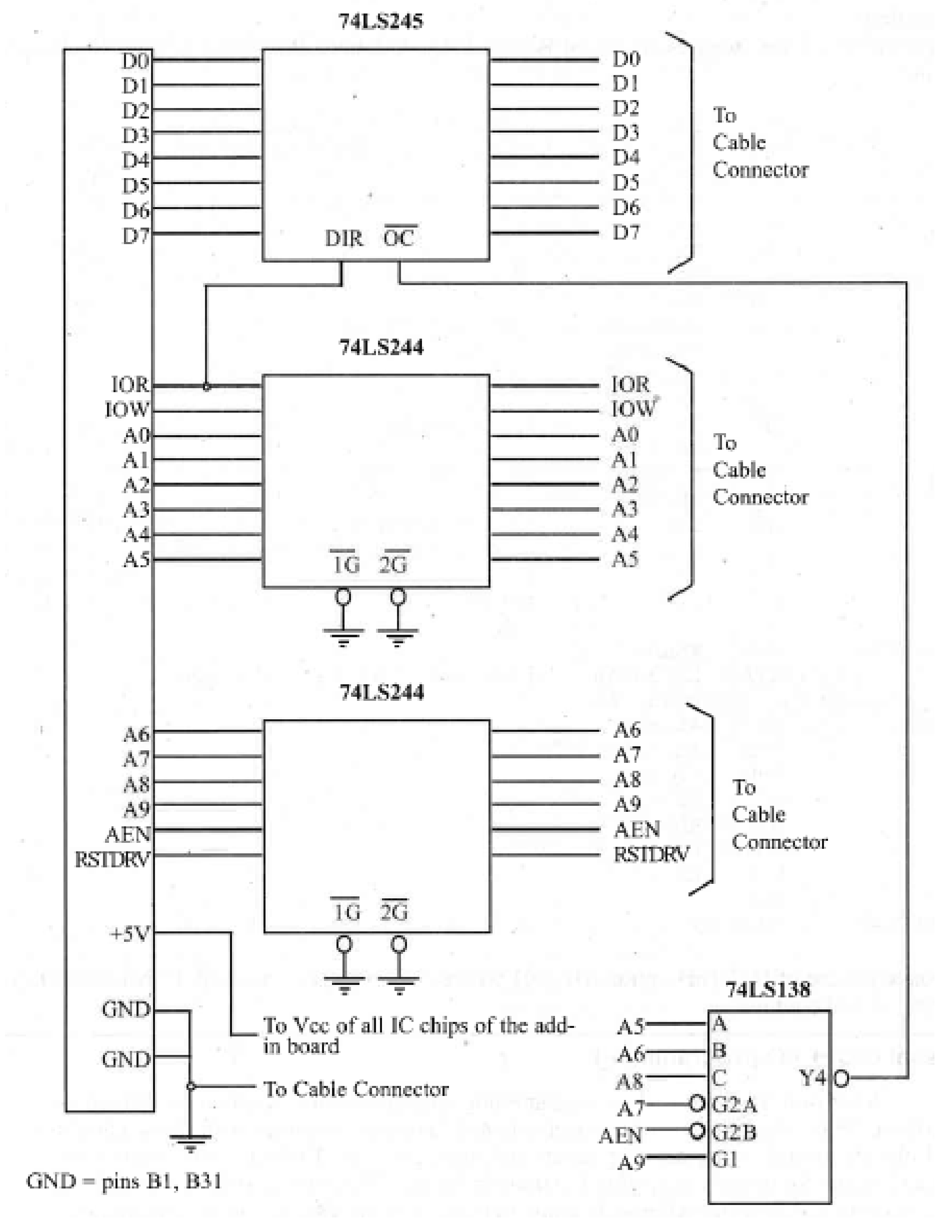
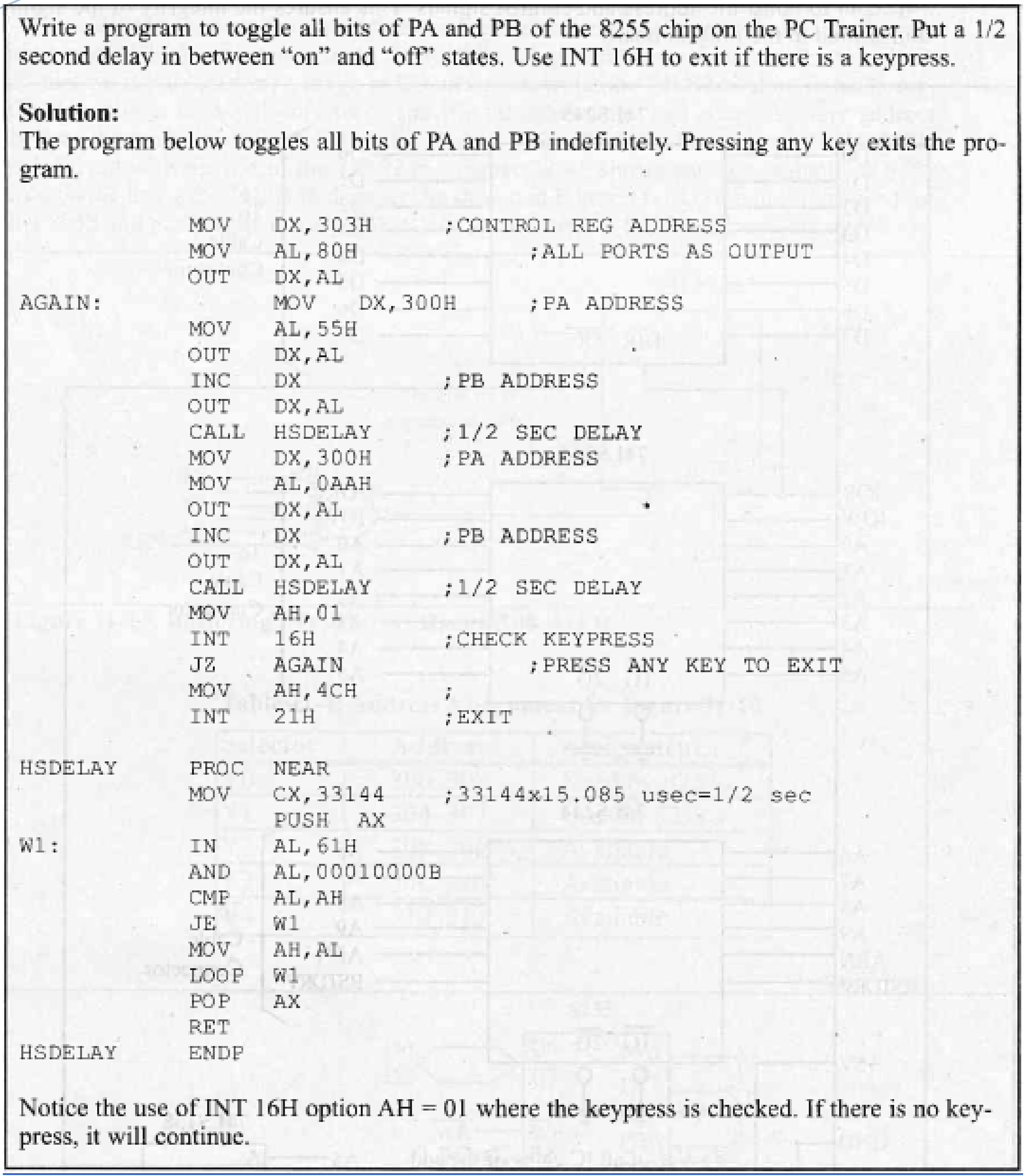


Fig: Design of 8-bit ISA PC Bus Extender

» The following shows a test program to toggle the PA and PB bits. Notice that in order to avoid locking up the system, INT 16H is used to exit upon pressing any key.



Write a program to toggle all bits of PA and PB of the 8255 chip on the PC Trainer. Put a 1/2 second delay in between "on" and "off" states. Use INT 16H to exit if there is a keypress.

Solution:
The program below toggles all bits of PA and PB indefinitely. Pressing any key exits the program.

```
            MOV     DX,303H      ;CONTROL REG ADDRESS
            MOV     AL,80H             ;ALL PORTS AS OUTPUT
            OUT     DX,AL
AGAIN:              MOV    DX,300H     ;PA ADDRESS
            MOV     AL,55H
            OUT     DX,AL
            INC     DX           ;PB ADDRESS
            OUT     DX,AL
            CALL    HSDELAY      ;1/2 SEC DELAY
            MOV     DX,300H      ;PA ADDRESS
            MOV     AL,0AAH
            OUT     DX,AL
            INC     DX           ;PB ADDRESS
            OUT     DX,AL
            CALL    HSDELAY      ;1/2 SEC DELAY
            MOV     AH,01
            INT     16H          ;CHECK KEYPRESS
            JZ      AGAIN              ;PRESS ANY KEY TO EXIT
            MOV     AH,4CH       ;
            INT     21H          ;EXIT

HSDELAY     PROC    NEAR
            MOV     CX,33144     ;33144x15.085 usec=1/2 sec
                    PUSH   AX
W1:         IN      AL,61H
            AND     AL,00010000B
            CMP     AL,AH
            JE      W1
            MOV     AH,AL
            LOOP    W1
            POP     AX
            RET
HSDELAY     ENDP
```

Notice the use of INT 16H option AH = 01 where the keypress is checked. If there is no keypress, it will continue.

Visual C/C++ I/O Programming:

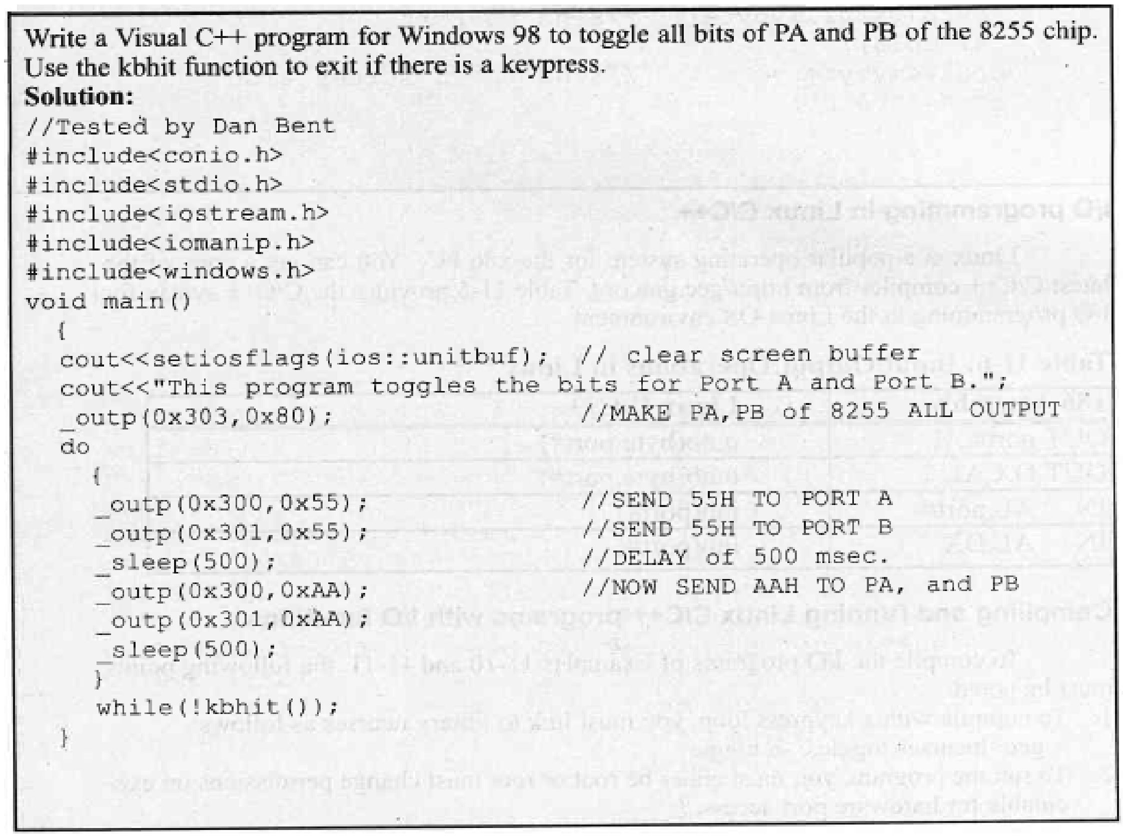
- Microsoft Visual C++ is a programming language widely used on the Windows platform.
- Since Visual C++ is an object-oriented language, it comes with many classes and objects to make programming easier and more efficient.
- But, there is no object or class for directly accessing I/O ports in the full Windows version of Visual C++.
- The reason for that is that Microsoft wants to make sure the x86 system programming is under full control of the operating system. This prevents any hacking into the system hardware.
- This applies to Windows NT, 2000, XP, and higher.

- o Hence, none of the system INT instructions such as INT 21H and I/O operations are applicable in Windows XP and its subsequent versions.
- o To access the I/O and other hardware features of the x86 PC in the XP environment you must use the Windows Platform SDK provided by Microsoft.

- The situation is different in the Windows 9x (95 and 98) environment.
- While INT 21H and other system interrupt instructions are blocked in Windows 9x, direct I/O addressing is available.
- To access I/O directly in Windows 9x, you must program Visual C++ in console mode.
- The instruction syntax for I/O operations is shown in the following Table.

x86 Assembly	Visual C++
OUT port#, AL	_outp (port#, byte)
OUT DX, AL	_outp (port#, byte)
IN AL, port#	_inp (port#)
IN AL, DX	_inp (port#)

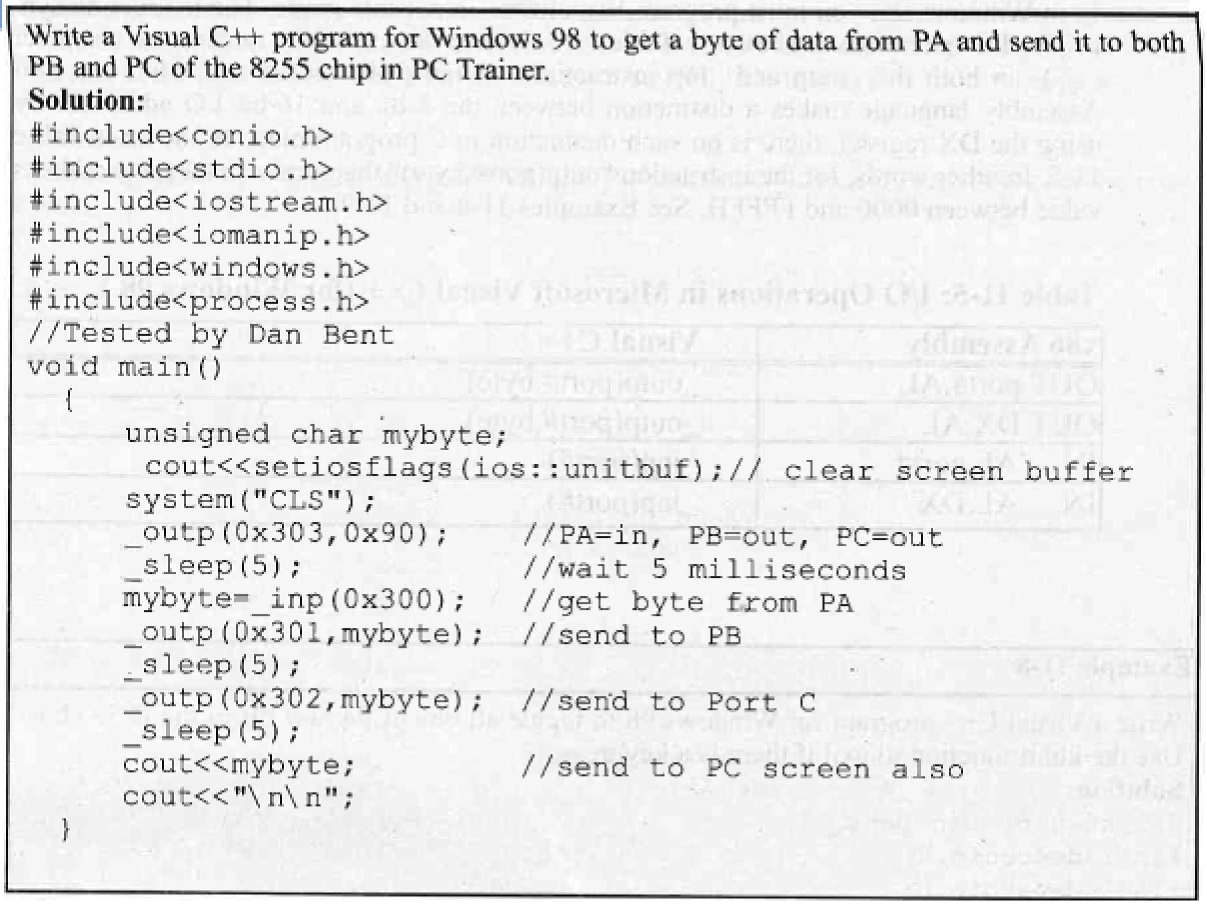
- Notice the use of the underscore character (_) in both the _outp and _inp instructions.
- Also note that, while the x86 Assembly language makes a distinction between the 8-bit and 16-bit I/O addresses by using the DX register, there is no such distinction in C programming. In other words, for the instruction "*outp (port#, byte)*" the port# can take any address value between 0000 and FFFFH.



Write a Visual C++ program for Windows 98 to toggle all bits of PA and PB of the 8255 chip. Use the kbhit function to exit if there is a keypress.

Solution:

```
//Tested by Dan Bent
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
#include<iomanip.h>
#include<windows.h>
void main()
  {
  cout<<setiosflags(ios::unitbuf);  // clear screen buffer
  cout<<"This program toggles the bits for Port A and Port B.";
  _outp(0x303,0x80);                //MAKE PA,PB of 8255 ALL OUTPUT
  do
    {
    _outp(0x300,0x55);              //SEND 55H TO PORT A
    _outp(0x301,0x55);              //SEND 55H TO PORT B
    _sleep(500);                    //DELAY of 500 msec.
    _outp(0x300,0xAA);              //NOW SEND AAH TO PA, and PB
    _outp(0x301,0xAA);
    _sleep(500);
    }
    while(!kbhit());
  }
```



Write a Visual C++ program for Windows 98 to get a byte of data from PA and send it to both PB and PC of the 8255 chip in PC Trainer.

Solution:

```
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
#include<iomanip.h>
#include<windows.h>
#include<process.h>
//Tested by Dan Bent
void main()
  {
     unsigned char mybyte;
      cout<<setiosflags(ios::unitbuf);// clear screen buffer
     system("CLS");
     _outp(0x303,0x90);     //PA=in, PB=out, PC=out
     _sleep(5);             //wait 5 milliseconds
     mybyte=_inp(0x300);    //get byte from PA
     _outp(0x301,mybyte);   //send to PB
     _sleep(5);
     _outp(0x302,mybyte);   //send to Port C
     _sleep(5);
     cout<<mybyte;          //send to PC screen also
     cout<<"\n\n";
  }
```

I/O Programming in Linux C/C++:

- Linux is a popular operating system for the x86 PC.
- The following Table provides the C/C++ syntax for I/O programming in the Linux OS environment.

x86 Assembly	Linux C/C++
OUT port#, AL	outb (byte, port#)
OUT DX, AL	outb (byte, port#)
IN AL, port#	inb (port#)
IN AL, DX	inb (port#)

Compiling & Running Linux C/C++ Programs with I/O Functions:

- To compile the I/O programs, the following points must be noted:
  - To compile with a keypress loop, you must link to library ncurses as follows:

```
> gcc -lncurses toggle.c -o toggle
```

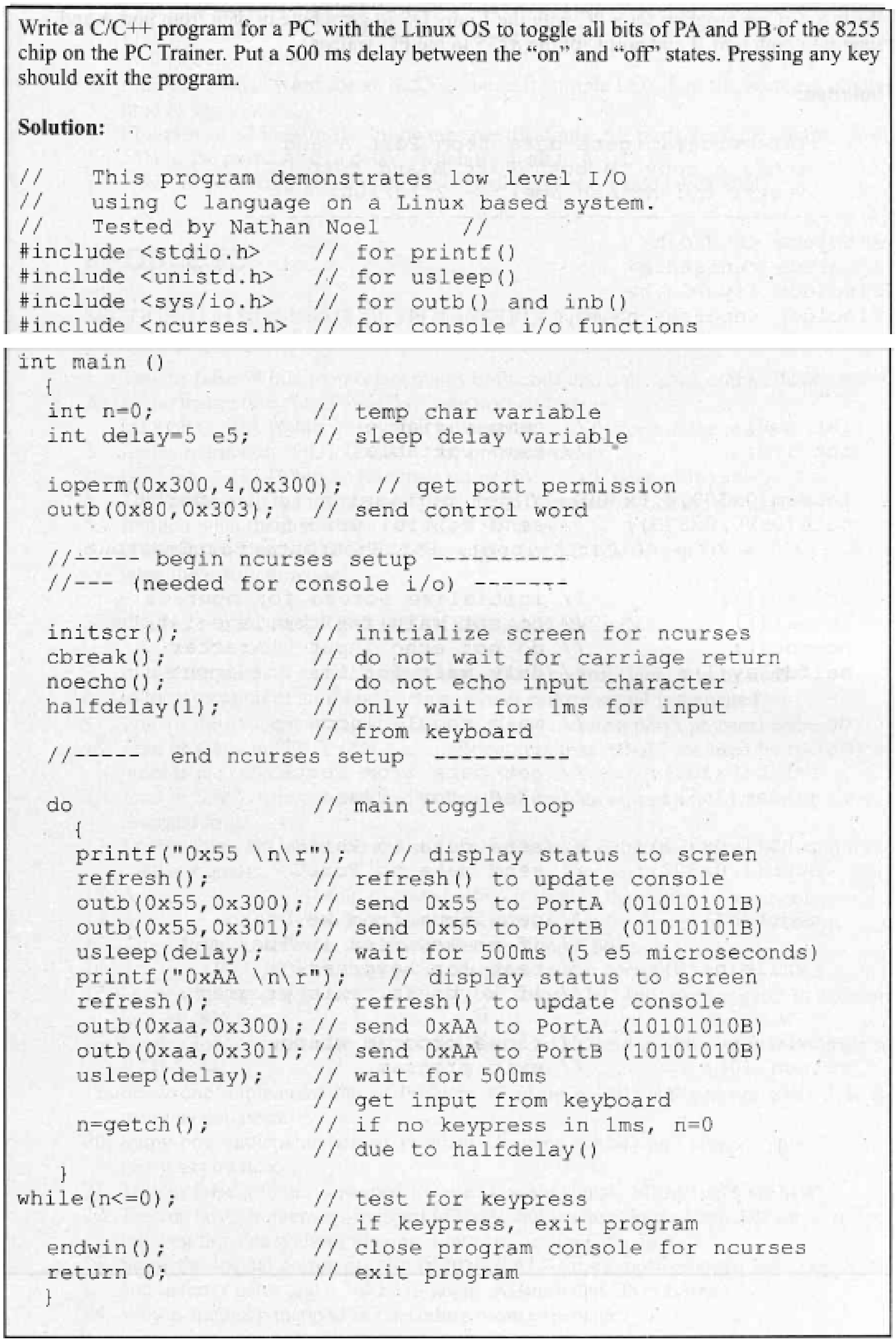
- To run the program, you must either be root or root must change permissions on executable for hardware port access.

  Example: (as root or superuser)

> chown root toggle

> chmod 4750 toggle

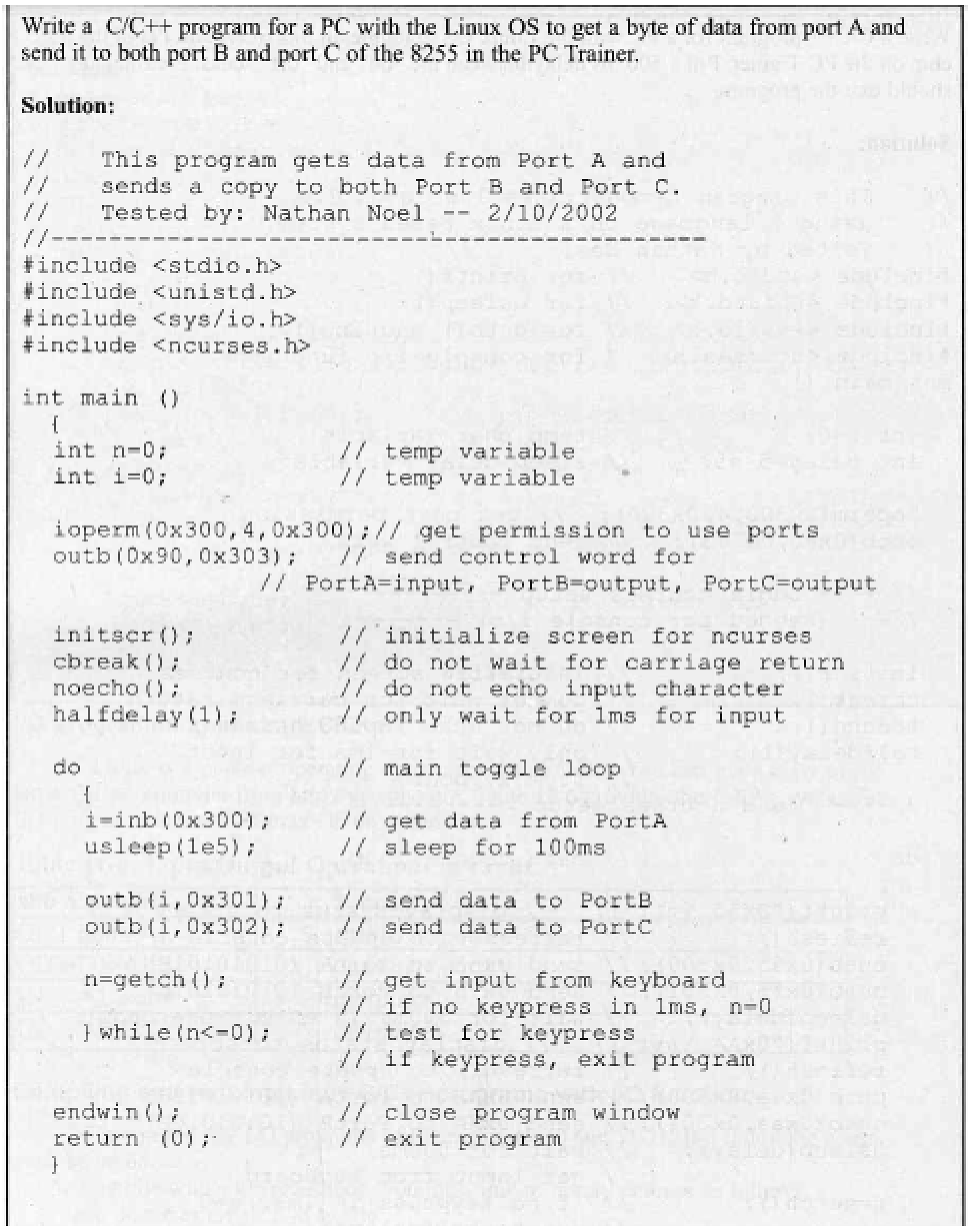
- Now toggle can be executed by users other than root.



Write a C/C++ program for a PC with the Linux OS to toggle all bits of PA and PB of the 8255 chip on the PC Trainer. Put a 500 ms delay between the "on" and "off" states. Pressing any key should exit the program.

Solution:

```
//    This program demonstrates low level I/O
//    using C language on a Linux based system.
//    Tested by Nathan Noel        //
#include <stdio.h>     // for printf()
#include <unistd.h>    // for usleep()
#include <sys/io.h>    // for outb() and inb()
#include <ncurses.h>   // for console i/o functions
int main  ()
  {
  int n=0;             // temp char variable
  int delay=5 e5;      // sleep delay variable

  ioperm(0x300,4,0x300);  // get port permission
  outb(0x80,0x303);    // send control word

  //----- begin ncurses setup -----------
  //--- (needed for console i/o) -------

  initscr();           // initialize screen for ncurses
  cbreak();            // do not wait for carriage return
  noecho();            // do not echo input character
  halfdelay(1);        // only wait for 1ms for input
                       // from keyboard
  //-----  end ncurses setup  ----------

  do                   // main toggle loop
    {
    printf("0x55 \n\r");    // display status to screen
    refresh();         // refresh() to update console
    outb(0x55,0x300); // send 0x55 to PortA (01010101B)
    outb(0x55,0x301); // send 0x55 to PortB (01010101B)
    usleep(delay);     // wait for 500ms (5 e5 microseconds)
    printf("0xAA \n\r");    // display status to screen
    refresh();         // refresh() to update console
    outb(0xaa,0x300); // send 0xAA to PortA (10101010B)
    outb(0xaa,0x301); // send 0xAA to PortB (10101010B)
    usleep(delay);     // wait for 500ms
                       // get input from keyboard
    n=getch();         // if no keypress in 1ms, n=0
                       // due to halfdelay()
   }
while(n<=0);           // test for keypress
                       // if keypress, exit program
  endwin();            // close program console for ncurses
  return 0;            // exit program
  }
```



Write a C/C++ program for a PC with the Linux OS to get a byte of data from port A and send it to both port B and port C of the 8255 in the PC Trainer.

Solution:

```
//    This program gets data from Port A and
//    sends a copy to both Port B and Port C.
//    Tested by: Nathan Noel -- 2/10/2002
//--------------------------------------------
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h>
#include <ncurses.h>

int main ()
  {
  int n=0;                // temp variable
  int i=0;                // temp variable

  ioperm(0x300,4,0x300);// get permission to use ports
  outb(0x90,0x303);     // send control word for
                   // PortA=input, PortB=output, PortC=output

  initscr();            // initialize screen for ncurses
  cbreak();             // do not wait for carriage return
  noecho();             // do not echo input character
  halfdelay(1);         // only wait for 1ms for input

  do                    // main toggle loop
    {
    i=inb(0x300);       // get data from PortA
    usleep(1e5);        // sleep for 100ms

    outb(i,0x301);      // send data to PortB
    outb(i,0x302);      // send data to PortC

    n=getch();          // get input from keyboard
                        // if no keypress in 1ms, n=0
    }while(n<=0);       // test for keypress
                        // if keypress, exit program

  endwin();             // close program window
  return (0);           // exit program
  }
```

By: MAHESH PRASANNA K.,

DEPT. OF CSE, VCET.

****

****