ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)
Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.
Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.
A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.
Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Computer Science and Engineering

Academic Year 2023-24

II. B.Tech I Semster

Digital Electronics & Microprocessors (20APC0503/ 20APC3601)

Prepared By

Ms. Deveswari
Assistant Professor
Department of ECE, AITS

UNIT - 1

Number System:

Number system is a basis for counting various items. On hearing the word 'number', all of us immediately think of the familiar decimal number system with its 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Modern computers communicate and operate with binary numbers which use only the digits 0 and 1. For large decimal quantities are deal with very large binary strings and therefore they do not like working with binary numbers. This gave rise to three new number systems.

- Octal
- Hexadecimal
- Binary Coded Decimal (BCD)

→ To define any number system, we have to specify following aspects:

* Base of the number system such as 2, 8, 10 or 16.
* The base (or) radix decides the total number of digits available in that number system.
* First digit in the number system is always zero (0) and last digit in the number system is always. base - 1.
* In general a number in a system having base can be written as

$$a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_0 \times r^0 + a_{-1} \times r^{-1} + a_{-2} \times r^{-2} + \dots + a_{-m} \times r^{-m}$$

Where a_n = the value of the nth digit, r = radix.

Decimal Number System:

- The decimal number system contain ten unique symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.
- In decimal system 10 symbols are involved, so the base or radix is 10.
- It is a weighted number system.
- The value attached to the symbol depends on its location with respect to the decimal point.

In general, $d_n d_{n-1} d_{n-2} \cdots d_0 \cdot d_{-1} d_{-2} \cdots d_{-m}$ is given by

$$(d_n \times 10^n) + (d_{n-1} \times 10^{n-1}) + \cdots + (d_0 \times 10^0) + (d_{-1} \times 10^{-1}) + (d_{-2} \times 10^{-2}) + \cdots + (d_{-m} \times 10^{-m}).$$

ex:- $9256.26 = 9\times10^3 + 2\times10^2 + 5\times10^1 + 6\times10^0 + 2\times10^{-1} + 6\times10^{-2}$.

$= 9\times1000 + 2\times100 + 5\times10 + 6 + (2/10) + (6/100)$.

Binary Number System:-

- The binary number system is a weighted system.
- The base of this number system is 2.
- It has two independent symbols.
- The symbols used are 0 and 1.
- A binary digit is called a bit.

Ex:- 1101.101

$\Rightarrow 1\times2^3 + 1\times2^2 + 0\times2^1 + 1 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

$\Rightarrow 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125$

$\Rightarrow (13.625)_{10}$

Octal Number System

- It is also a weighted system.
- Its base or radix is 8.
- It has 8 independent symbols 0, 1, 2, 3, 4, 5, 6 and 7.
- Its base $8 = 2^3$, every 3-bit group of binary can be represented by an octal digit.

Ex:- $(567)_8$

Hexa Decimal Number System:

- The hexadecimal number system is a weighted system.
- The base or radix of this number system is 16.
- The symbols used are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.
- The base $16 = 2^4$, every 4-bit group of binary can be represented by an hexadecimal digit.

Ex:- $(3FD)_{16}$.

Conversions:

- Decimal to binary conversion:-

(i) Convert $(52)_{10}$ into binary

Sol:-

```
2 | 52
2 | 26  - 0
2 | 13  - 0   ↑
2 | 6   - 1
2 | 3   - 0
    1   - 1
```

$(52)_{10} = (110100)_2$

(ii) Convert $(105.15)_{10}$ into binary

Integer part

$2 \,|\, 105$
$2 \,|\, 52 - 1$
$2 \,|\, 26 - 0$
$2 \,|\, 13 - 0$
$2 \,|\, 6 - 1$
$2 \,|\, 3 - 0$
$1 - 1$

Fraction part

$0.15 \times 2 = 0.30 \rightarrow 0$
$0.30 \times 2 = 0.60 \rightarrow 0$
$0.60 \times 2 = 1.20 \rightarrow 1$
$0.20 \times 2 = 0.40 \rightarrow 0$
$0.40 \times 2 = 0.80 \rightarrow 0$
$0.80 \times 2 = 1.60 \rightarrow 1$

$(105.15)_{10} = (1101001.001001)_2$

- Decimal to octal Conversion:

(i) Convert $(378.93)_{10}$ into octal.

$8 \,|\, 378$
$8 \,|\, 47 - 2$
$5 - 7$

$0.93 \times 8 = 7.44 \rightarrow 7$
$0.44 \times 8 = 3.52 \rightarrow 3$
$0.52 \times 8 = 4.16 \rightarrow 4$
$0.16 \times 8 = 1.28 \rightarrow 1$

$(378.93)_{10} = (572.7341)_8$

- Decimal to hexadecimal conversion:

Convert $(2598.675)_{10}$ into hexadecimal.

$16 \,|\, 2598$
$16 \,|\, 162 - 6$
$10 - 2$

$0.675 \times 16 = 10.8 \rightarrow A$
$0.8 \times 16 = 12.8 \rightarrow C$
$0.8 \times 16 = 12.8 \rightarrow C$
$0.8 \times 16 = 12.8 \rightarrow C$

$(2598.675)_{10} = (A26.ACCC)_{16}$

Binary to Decimal Conversion

i, Convert $(10101)_2$ to decimal

$$(10101)_2 = (1\times2^4)+(0\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)$$

$$= 16+0+4+0+1$$

$$= (21)_{10}$$

ii, Convert $(111.101)_2$ to decimal

$$(111.101)_2 = (1\times2^2)+(1\times2^1)+(1\times2^0)+(1\times2^{-1})+(0\times2^{-2})+(1\times2^{-3})$$

$$= 4+2+1+0.5+0+0.125$$

$$= 7.625$$

$$\Rightarrow (111.101)_2 = (7.625)_{10}$$

Octal to Decimal Conversion

Convert $(756.603)_8$ to decimal

$$(756.603)_8 = (7\times8^2)+(5\times8^1)+(6\times8^0)+(6\times8^{-1})+(0\times8^{-2})+(3\times8^{-3})$$

$$= 448+40+6+0.75+0+0.005$$

$$= (494.755)_{10}$$

Hexadecimal to decimal Conversion

Convert $(A0F9.0EB)_{16}$ to decimal

$$(A0F9.0EB)_{16} = (10\times16^3)+(0\times16^2)+(15\times16^1)+(9\times16^0)$$
$$+(0\times16^{-1})+(14\times16^{-2})+(11\times16^{-3})$$

$$= 40960+0+240+9+0+0.0546+0.0026$$

$$= (41209.0572)_{10}$$

Binary to octal Conversion

for binary to octal conversion the binary numbers are divided into groups of 3 bits each.

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

ii, Convert $(101111010110 \cdot 110110011)_2$ into octal.

$$\underbrace{101}_{5}\,\underbrace{111}_{7}\,\underbrace{010}_{2}\,\underbrace{110}_{6} \cdot \underbrace{110}_{6}\,\underbrace{110}_{6}\,\underbrace{011}_{3}$$

$\Rightarrow (101111010110 \cdot 110110011)_2 = (5726 \cdot 663)_8$

iii, Convert $(10101111001 \cdot 0111)_2$ into octal.

$$\underbrace{010}_{2}\,\underbrace{101}_{5}\,\underbrace{111}_{7}\,\underbrace{001}_{1} \cdot \underbrace{011}_{3}\,\underbrace{100}_{4}$$

$\Rightarrow (10101111001 \cdot 0111)_2 = (2571 \cdot 34)_8$

Binary to Hexadecimal conversion:

for binary to hexadecimal conversion the binary numbers are divided into groups of 4 bits each.

Hexadecimal	Binary	Hexadecimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

i) Convert $(1011011011)_2$ into hexadecimal.

$$\underline{0010}\,\underline{1101}\,\underline{1011}$$

$$2 \quad D \quad B$$

$\Rightarrow (1011011011)_2 = (2DB)_{16}$

ii) Convert $(01011111011 \cdot 011111)_2$ into hexadecimal.

$$\underline{0010}\;\underline{1111}\;\underline{1011} \cdot \underline{0111}\;\underline{1100}$$

$$2 \quad F \quad B \;\cdot\; 7 \quad C$$

$\Rightarrow (01011111011 \cdot 011111)_2 = (2FB \cdot 7C)_{16}$

Octal to Binary Conversion:

To Convert octal number to binary, replace each octal digit by its 3-bit binary equivalent.

Convert $(367.52)_8$ into binary

3 6 7 . 5 2

011 110 111 . 101 010

$\Rightarrow (367.52)_8 = (011110111.101010)_2$

Hexadecimal to Binary Conversion:

To Convert Hexadecimal number to binary, replace each hexadecimal digit by its 4-bit binary group.

Convert $(3A9E.B0D)_{16}$ into binary

$(3A9E.B0D)_{16}$

0011 1010 1001 1110 . 1011 0000 1101

$\Rightarrow (3A9E.B0D)_{16} = (0011101010011110.101100001101)_2$

Octal to hexadecimal Conversion:

For octal to Hexadecimal conversion, first convert the given octal number to binary and then binary number to hexadecimal.

Convert $(756.603)_8$ to hexadecimal

7 5 6 . 6 0 3

111 101 110 . 110 000 011

$\underbrace{0001}_{1}\ \underbrace{1110}_{E}\ \underbrace{1110}_{E}\ .\ \underbrace{1100}_{C}\ \underbrace{0001}_{1}\ \underbrace{1000}_{8}$

$\Rightarrow (756 \cdot 603)_8 = (1EE \cdot C18)_{16}$

Hexadecimal to Octal Converion

For hexadecimal to Octal Converion, first convert the given hexadecimal number to binary and then binary number to octal.

Convert $(B9F \cdot AE)_{16}$ to octal.

```
 B    9    F   .  A    E
1011 1001 1111 . 1010 1110

101 110 011 111 . 101 011 100
 5   6   3   7  .  5   3   4
```

$\Rightarrow (B9F \cdot AE)_{16} = (5637 \cdot 534)_8$

Binary Arithematic Operation:

1. Binary Addition:

0 + 0 = 0,
0 + 1 = 1,
1 + 0 = 1,
1 + 1 = 10, i.e. 0 with carry 1

* Add $(100101)_2$ and $(1101111)_2$

```
   100101
+ 1101111
  1 1111
---------
 10010100
---------
```

2. Binary Subtraction:

0 - 0 = 0

0 - 1 = 1, with a borrow of 1

1 - 0 = 1

1 - 1 = 0

Substract $(111.111)_2$ from $(1010.01)_2$

```
  1010.010
   111.111
 ----------
  0010.011
 ----------
```

3. Binary Multiplication:

0×0 = 0

0×1 = 0

1×0 = 0

1×1 = 1

Multiply $(1101)_2$ by $(110)_2$

```
      1101
    ×  110
   -------
      0000
     1101
    1101
  --------
   1001110
  --------
```

4. Binary Division :

$0 \div 1 = 0$

$1 \div 1 = 1$

Divide $(101101)_2$ by $(110)_2$

```
110) 101101 (111·1
     110
     ----
     1010
      110
     -----
      1001
       110
      -----
        110
        110
       -----
        000
       -----
```

Representation of Signed no.s.

Two ways of representing signed no.s

- Sign Magnitude form
- Complemented form

→ In Sign magnitude form, an additional bit called the sign bit is placed in front of the no. If the sign bit is 0, the no. is +ve, if it is 1, the no. is -ve

Ex:- 0 110100 = +52
↓ Sign bit

1 110100 = -52

↓

sign bit

* Complemented Form

- 1's Complement
- 2's Complement

→ 1's Complement Representation

The 1's Complement of a binary number is obtained by changing each 0 to 1 and each 1 to 0.

Ex:- Find $(1100)_2$ 1's Complement

1 1 0 0

0 0 1 1 → 1's Complement

→ 2's Complement Representation

The 2's complement of a binary number is a binary number which is obtained by adding 1 to the 1's complement of a number i.e

2's complement = 1's complement +1

Ex:- Find $(1010)_2$ 2's Complement

1 0 1 0

0 1 0 1 → 1's Complement

+ 1

0 1 1 0 → 2's Complement

- perform $(1011)_2 - (0100)_2$ using 1's complement method.

$$\begin{array}{rcll} 11 & \rightarrow & 1011 & = A \\ -\ 4 & \rightarrow & 0100 & = B \\ \hline 7 & & & \end{array}$$

$A > B$

1's complement for 0100 is 1011

$$\begin{array}{r} 1011 \\ +\ 1011 \\ \hline \boxed{1}0110 \\ \hookrightarrow 1 \\ \hline 0111 \\ \hline \end{array}$$

- Subtract $(9)_{10}$ from $(4)_{10}$ using 1's complement method

$A = (4)_{10} = (0100)_2$ $A < B$

$B = (9)_{10} = (1001)_2$

1's complement for 1001 is 0110

$$\begin{array}{r} 0100 \\ +\ 0110 \\ \hline 1010 \\ \hline \end{array}$$

Perform $(9)_{10} - (5)_{10}$ using 2's Complement method.

$A = (9)_{10} = (1001)_2$

$B = (5)_{10} = (0101)_2$

1's complement for 0101 is 1010

$$\begin{array}{r} 1010 \\ +\quad 1 \\ \hline 1011 \\ \hline \end{array}$$

2's Complement ← 1011

$$\begin{array}{r} 1001 \\ 1011 \\ \hline ①1100 \end{array}$$

↳ Discard carry.

perform $(4)_{10} - (9)_{10}$ using the 2's Complement method.

A < B

$A = (4)_{10} = (0100)_2$

$B = (9)_{10} = (1001)_2$

2's complement for 1001 is 0111

$$\begin{array}{r} 0100 \\ 0111 \\ \hline 1011 \\ \hline \end{array}$$

Add -45.75 to +87.5 using 12 bit arithmetic

+87.5 = 01010111.1000

-45.75 = 11010010.0100 → 2's Complement

$$\begin{array}{r} \textcircled{1}00101001.1100 \end{array}$$

← Discard carry

9's Complement & 10's Complement

* 9's complement of 3465 and 782.54

$$\begin{array}{r} 9999 \\ -3465 \\ \hline 6534 \end{array} \qquad \begin{array}{r} 999.99 \\ -782.54 \\ \hline 217.45 \end{array}$$

* 10's Complement of 4069 is

$$\begin{array}{r} 9999 \\ -4069 \\ \hline 5930 \\ +1 \\ \hline 5931 \end{array}$$

5930 → 9's complement

5931 → 10's complement

· Subtract 745.81 - 436.62 using 9's complement

$$\begin{array}{r} 745.81 \\ -436.62 \\ \hline 309.19 \end{array} \qquad \begin{array}{r} 745.81 \\ +563.37 \\ \hline \textcircled{1}309.18 \\ \xrightarrow{} 1 \\ \hline 309.19 \end{array} \qquad \begin{array}{r} 999.99 \\ -436.62 \\ \hline 563.37 \end{array}$$

563.37 → 9's Complement

- Subtract using 9's complement 436·62 − 745·82

$$\begin{array}{r} 436{\cdot}62 \\ -\ 745{\cdot}82 \\ \hline -\ 309{\cdot}20 \\ \hline \end{array}$$

9's complement for 745·82 is

$$\begin{array}{r} 999{\cdot}99 \\ -\ 745{\cdot}82 \\ \hline 254{\cdot}17 \end{array}$$

$$\begin{array}{r} 436{\cdot}62 \\ +\ 254{\cdot}17 \\ \hline 690{\cdot}79 \end{array}$$

If no carry result in -ve

If it has carry it is +ve

Result in 9's complement 690·79

$$\rightarrow \begin{array}{r} 999{\cdot}99 \\ 690{\cdot}79 \\ \hline -\ 309{\cdot}20 \\ \hline \end{array}$$

- Subtract 2928·54 − 416·73 using 10's complement

$$\begin{array}{r} 2928{\cdot}54 \\ -\ 0416{\cdot}73 \\ \hline 2511{\cdot}81 \\ \hline \end{array}$$

9's complement for 416·73

$$\begin{array}{r} 999{\cdot}99 \\ -\ 0416{\cdot}73 \\ \hline 9583{\cdot}26 \\ +1 \\ \hline 9583{\cdot}27 \end{array}$$

↓

10's compleme

$$\begin{array}{r} 2928{\cdot}54 \\ +\ 9583{\cdot}27 \\ \hline ①2511{\cdot}81 \end{array}$$

↳ Ignore carry

Subtract 416.73 - 2928.54 using 10's complement

$$\begin{array}{r} 0416.73 \\ -\ 2928.54 \\ \hline -2511.81 \\ \hline \end{array}$$

10's complement for 2928.54 is

$$\begin{array}{rl} 9999.99 & \\ -\ 2928.54 & \\ \hline 7071.45 & \rightarrow 9\text{'s comp} \\ +1 & \\ \hline 7071.46 & \rightarrow 10\text{'s comp} \end{array}$$

$$\begin{array}{r} 0416.73 \\ +\ 7071.46 \\ 1 \\ \hline 7488.19 \\ \hline \end{array}$$

Binary Coded Decimal (BCD):

BCD is a weighted codes, each successive digit from right to left represents weights equal to some specified value and to get the equivalent decimal number add the products of the weights by the corresponding binary digit. 8421 is the most common because 8421 BCD is the most natural amongst the other possible codes.

Decimal	BCD	Decimal	BCD
	8421		8421 8421
0	0000	10	0001 0000
1	0001	11	0001 0001
2	0010	12	0001 0010
3	0011	13	0001 0011
4	0100		
5	0101		
6	0110		
7	0111		
8	1000		
9	1001		

* BCD Addition :-

Case1 : Sum is equal or less than 9 and carry is 0.

perform BCD addition of $(2)_{10}$ and $(6)_{10}$

```
                BCD
(2)10  →       0010
(6)10  →     + 0110
               1 1
             ------
               1000  → sum is a valid
                            number.
```

Case 2 : Sum greater than 9 but carry = 0

perform BCD addition of $(7)_{10}$ and $(6)_{10}$

```
                  BCD
  7   →          0111
  6   →        + 0110
 ---              1 1
 13             ------
                 1101  → Invalid BCD
               + 0110  → add 6 to Invalid
                  1                  BCD
               ------
                10011

            0001 0011  → valid BCD.
            ---- ----
              1    3
```

Case 3: Sum less than or equal to 9 but carry = 1

Perform BCD addition of $(9)_{10}$ and $(8)_{10}$.

```
   9    →      1001
 + 8    →    + 1000
 ---         ------
  17         1 0001  → Valid BCD sum
                          and carry=1
             ↓   ↓
          0001  0001  → Incorrect BCD result
• Add +6 to
          0000  0110
Incorrect BCD
  result  ----------
          0001  0111  → Correct BCD
          ----  ----
            1     7
```

Add $(569)_{10}$ and $(687)_{10}$ in BCD

$$\begin{array}{r} 569 \\ +\ 687 \\ \hline 1256 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \\ \end{array} \quad \begin{array}{rrr} 0101 & 0110 & 1001 \\ +\ 0110 & 1000 & 0111 \\ \hline 1011 & 1111 & 0000 \end{array}$$

1011 ↓ Invalid BCD

1111 ↓ Invalid BCD

0000 ↳ valid BCD with carry 1

Add $(0110)_2$ to Invalid BCD number to get correct BCD.

$$\begin{array}{rrr} 1011 & 1111 & 0000 \\ +\ 0110 & 0110 & 0110 \\ \hline 1\ 0010 & 0101 & 0110 \end{array}$$

↓

$\underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{0101}_{5}\ \underbrace{0110}_{6}$ → Valid BCD

* BCD Subtraction:

Substract $(38)_{10}$ and $(15)_{10}$ in BCD

$$\begin{array}{r} 38 \\ -\ 15 \\ \hline 23 \end{array} \quad \begin{array}{c} \rightarrow \\ \rightarrow \\ \\ \end{array} \quad \begin{array}{rr} 0011 & 1000 \\ -\ 0001 & 0101 \\ \hline 0010 & 0011 \end{array}$$

Valid BCD

- $(206.7)_{10} - (147.8)_{10}$ in BCD

```
  206.7   →   0010 0000 0110 . 0111
- 147.8   →   0001 0100 0111 . 1000
 ------      ----------------------
   58.9       0000 1011 1110 . 1111
                        0110 0110 . 0110
              ----------------------
Valid →                0101 1000 . 1001
BCD                     ↓    ↓      ↓
                        5    8  .   9
```

borrow from the next group

Subtract with $(0110)_2$

- perform $(83)_{10} - (21)_{10}$ using 9's complement method

9's complement for 21 is

```
 99
-21
---
 78
```

```
83   →    1000 0011
78   →  + 0111 1000
         ----------
          1111 1011
           ↓     ↓
       Invalid  Invalid
         BCD     BCD
```

```
     1111 1011
   + 0110 0110
   -----------
 (1) 0110 0001
  ↳ carry
    add
    to
    LSB
```

that is

```
   0110 0001
 +         1
 -----------
   01100010
      ↓
  Correct BCD
```

perform $(54)_{10} - (22)_{10}$ in BCD using 10's complement method.

```
10's complement for 22 is  99
                           22
                          ----
                           77, → 9's comp
                           +1
                          ----
                           78 → 10's comp
```

```
                  BCD

54   →   0101  0100

78   →  +0111  1000
          111
        ----------
         1100  1100    → Invalid BCD
        +0110  0110
         1     1
        ----------
Ignore ← 1 0011  0010    → Valid BCD
carry
```

Excess-3 (xs-3) code:

It is a non - weighted BCD code. Each binary codeword is the corresponding 8421 codeword plus 0011 (3). It is a sequential code & therefore can be used for arithmetic operation.

Decimal digit	Excess-3 Code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

* Excess-3 Addition

To perform Excess-3 addition, we have to

- Add two excess-3 number.
- If carry = 1 → add 3 to the sum of two digits
  = 0 → Subtract 3

Ex:-

```
                XS-3 code
 4     →         0111
+3     →       + 0110
                  11
----           ------
 7               1101 → carry is 0
               - 0011
               ------
                 1010
```

* Excess-3 Subtraction

To perform Excess-3 subtraction, we have to

- Complement the subtrahend
- Add complemented subtrahend to minuend.
- If carry = 1, Result is +ve. Add 3 and end around carry
- If carry = 0, Result is -ve, subtract 3.

Ex:- 8 - 5

```
         xs3 code
8 →      1011
5 →      1000                      1011
          ↓ 1's complement       + 0111
         0111                       111
                                 ------
                                ① 0010
                                  └──→1
                                 ------
                                   0011
                                   0011
                                 ------
                                   0110
```

perform the subtraction $(645)_{10} - (319)_{10}$ in XS-3 code using the 9's complement.

9's complement for 319 is

$$\begin{array}{r} 999 \\ 319 \\ \hline 680 \\ \hline \end{array}$$

XS-3 Code

$$\begin{array}{lllll}
645 & \rightarrow & 1001 & 0111 & 1000 \\
680 & \rightarrow & +\,1001 & 1011 & 0011 \\
 & & \quad 11 & 111 & \\
\hline
 & \boxed{1} & 0011 & 0010 & 1011 \\
 & & +\,0011 & +\,0011 & -\,0011 \\
 & & \quad 11 & \quad 1 & \\
\hline
 & & 0110 & 0101 & 1000 \\
 & & & & \rightarrow +1 \\
\hline
 & & 0110 & 0101 & 1001 \\
\hline
\end{array}$$

Gray Code

Gray code is a special case of unit-distance code. In unit-distance code, bit patterns for two consecutive numbers differ in only one bit position. These codes are also called cyclic codes.

* **BINARY - To - GRAY CONVERSION:-**

If an n-bit binary number is represented by $B_n, B_{n-1} \cdots \cdots B_1$ and its gray code equivalent by, $G_n, G_{n-1} \cdots G_1$. Where G_n and B_n are the MSBs. then gray code bits are obtained from the binary code as follows.

$$G_n = B_n$$

$$G_{n-1} = B_n \oplus B_{n-1}$$

$$\vdots$$

$$G_1 = B_2 \oplus B_1$$

Where the symbol $\oplus$ stands for Exclusive OR (X-OR)

Ex:- 1001 to Gray code

Binary → $1 \xrightarrow{\oplus} 0 \xrightarrow{\oplus} 0 \xrightarrow{\oplus} 1$

↓ ↓ ↓ ↓

Gray → 1 1 0 1

The gray code for $(1001)_2$ is 1101

Decimal code	Binary code	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

* GRAY - TO - BINARY CONVERSION:

If an n-bit gray number is represented by $G_n \, G_{n-1} \cdots G_1$ and its binary equivalent by $B_n \, B_{n-1} \cdots B_1$, then binary bits are obtained from Gray bits as follows.

$$B_n = G_n$$

$$B_{n-1} = B_n \oplus G_{n-1}$$

$$\vdots$$

$$B_1 = B_2 \oplus G_1$$

Ex:- Convert the gray code 1101 to binary

$$\begin{tikzcd}
\text{Gray} \rightarrow & 1 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] \\
\text{Binary} \longrightarrow & 1 \arrow[ur, "\oplus"] & 0 \arrow[ur, "\oplus"] & 0 \arrow[ur, "\oplus"] & 1
\end{tikzcd}$$

The binary code is 1001.

Logic Gates

- Logic gates are the fundamental building blocks of digita systems.

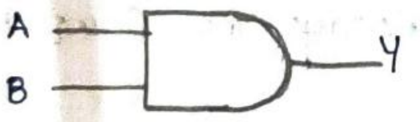
There are 3 basic types of gates AND, OR and NOT.

Logic gates are electronic circuits because they are made up of a number of electronic devices and components.

- Inputs and outputs of logic gates can occur only in 2 levels. These two levels are termed HIGH and LOW, or TRUE and FALSE, or ON and OFF or Simply 1 and 0.

AND Gate

- An AND gate has two or more inputs but only one output. The output is logic 1 state only when each one of its puts is at logic 1 state.
- the Output is logic 0 state even if one of its inputs is at logic 0 state



Logic Symbol

Truth Table

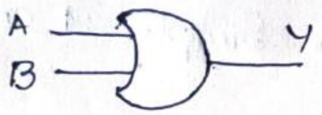
Inputs		output
A	B	Y = A·B
0	0	0
0	1	0
1	0	0
1	1	1

1	1	0

OR Gate:

- An OR gate may have two or more inputs but only one output.
- The output is logic 1 state, even if one of its input is in logic 1 state.
- The output is logic 0 state, only when each one of its inputs is in logic 1 state.

Logic Symbol



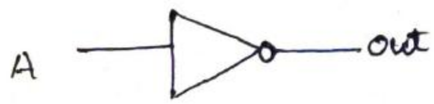
Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

- A NOT gate, also called an Inverter, has only one input and one output.
- It is a device whose output is always the Complement of its input.
- The output of a NOT gate is the logic 1 state when its input is in logic 0 state and the logic 0 state when its inputs is in logic 1 state.

Logic Symbol



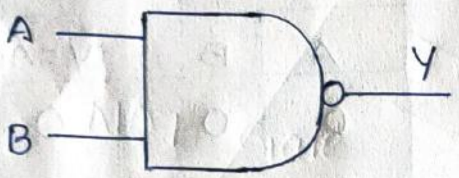
Truth Table

Input	Output
A	$\bar{A}$
0	1
1	0

NAND Gate

- NAND gate is a combination of an AND gate and a NOT gate
- The output is logic 0 when each of the input is logic 1 and for any other combination of inputs, the output is logic 1.

Logic Symbol



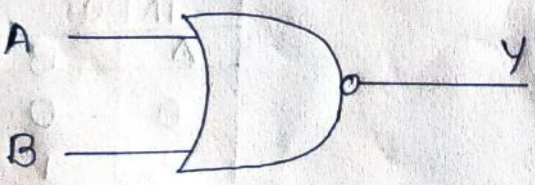
Truth Table

INPUT		OUTPUT
A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

- NOR gate is a combination of an OR gate and a NOT gate.
- The output is logic 1, only when each one of its input is logic 0 and for any other combination of inputs, the output is a logic 0 level.

Logic Symbol



Truth Table

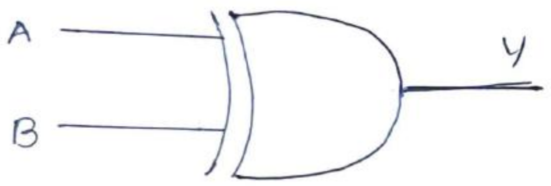
INPUT		OUTPUT
A	B	$Y = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

X -OR Gate

- An X- OR gate is a two input, one output logic circuit.

The output is logic 1 when one and only one of its two inputs is logic 1. When both the inputs is logic 0 or when both the inputs is logic 1, the output is logic 0.

Logic Symbol



$Y = A\bar{B} + \bar{A}B$

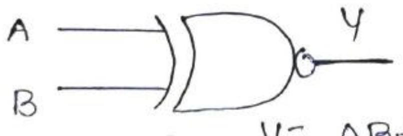
Truth Table

INPUT		OUTPUT
A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

X - NOR Gate

- An X- NOR gate is the combination of an X-OR gate and a NOT gate.
- An X- NOR gate is a two input, one output logic circuit.
- The output is logic 1 only when both the inputs are logic 0 or when both the inputs is 1.
- The output is logic 0 when one of itsputs is logic 0 and other is 1.

Logic Symbol



$Y = AB + \bar{A}\bar{B}$

Truth Table

INPUT		OUTPUT
A	B	$Y = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

Boolean Algebra

Boolean Algebra is used to analyze and simplify the digital circuits. Because, it uses only the binary numbers, i.e., 0 and 1, it is also called as Binary Algebra or Logical Algebra.

Boolean Laws

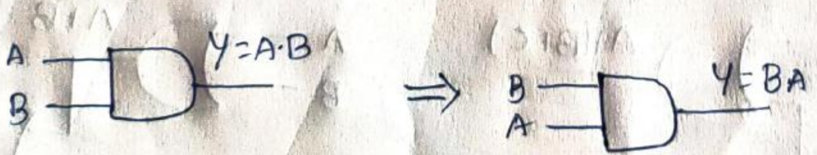
1. Commutative law
2. AND law
3. Associative law
4. Distributive law
5. OR law
6. Inversion law

Commutative Law

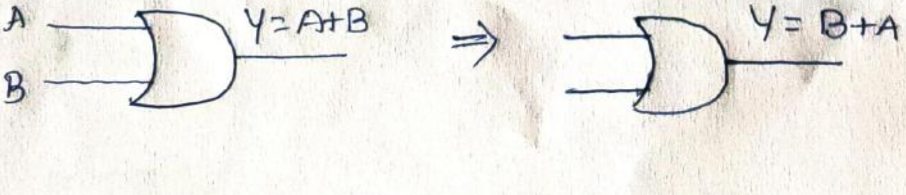
Any binary operation which satisfies the following expression is called as Commutative operation.

(i) $A \cdot B = B \cdot A$

(ii) $A + B = B + A$



(i)

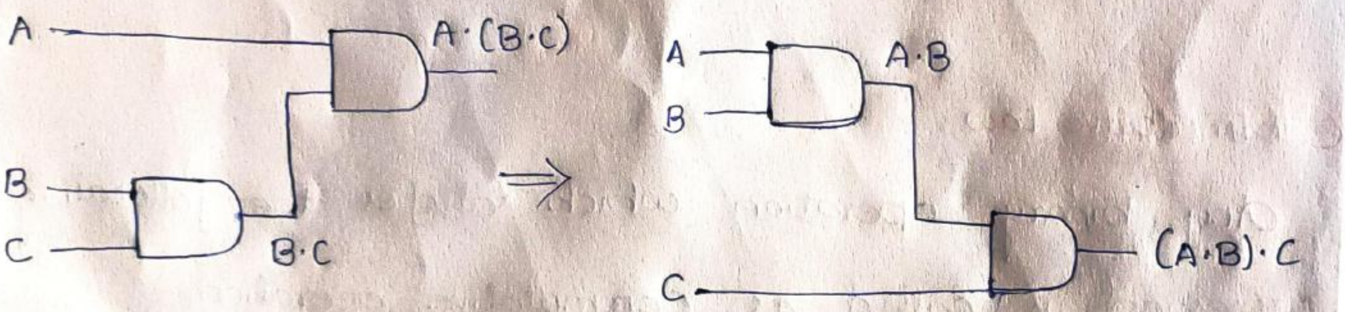


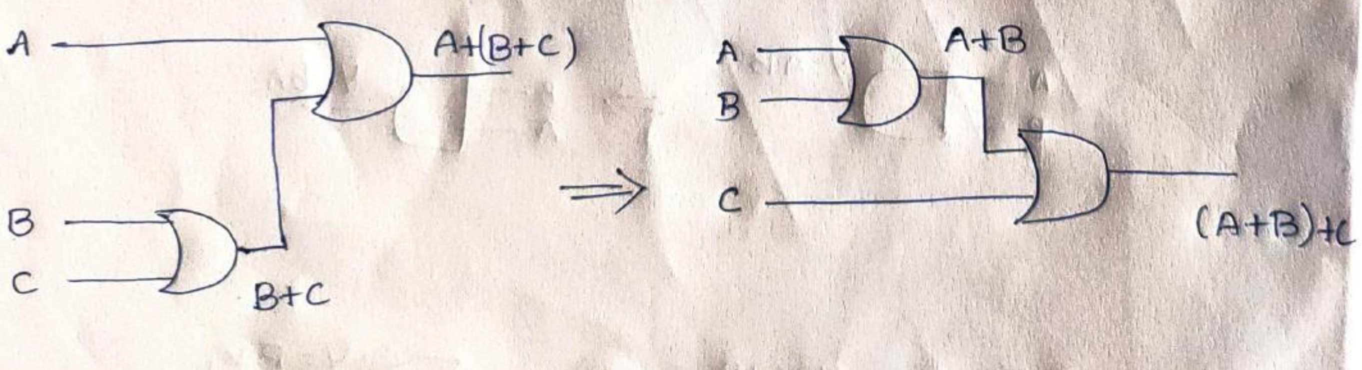
Inputs		A·B	B·A	A+B	B+A
A	B				
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1

Associative law

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$(A+B) + C = A + (B+C).$$

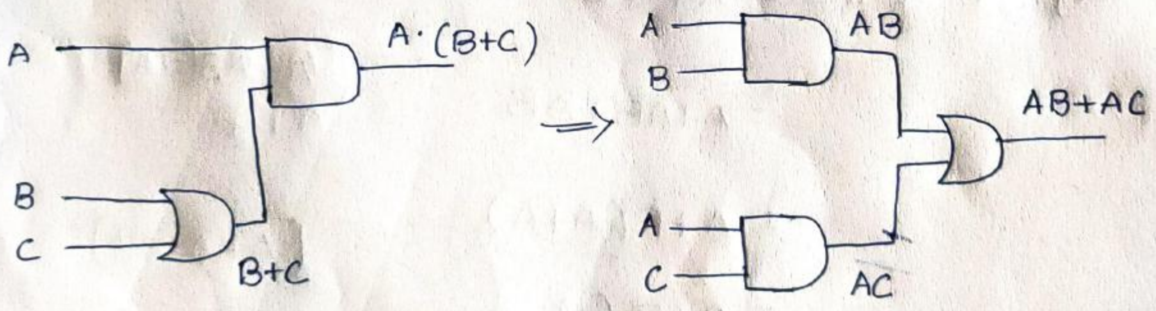




INPUTS			B·C	A·(B·C)	A·B	(A·B)·C
A	B	C				
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	0
1	1	1	1	1	1	1

Distributive Law:

$$A(B+C) = AB + AC$$



A	B	C	B+C	A·(B+C)	AB	AC	AB+AC
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

AND Laws

- * $A \cdot 0 = 0$
- * $A \cdot 1 = A$
- * $A \cdot A = A$
- * $A \cdot \bar{A} = 0$

OR Laws

- * $A + 0 = A$
- * $A + 1 = 1$
- * $A + A = A$
- * $A + \bar{A} = 1$

Inversion Law

- * If $A = 0$ then $\bar{A} = 1$, $\bar{\bar{A}} = 0$
- * If $A = 1$ then $\bar{A} = 0$, $\bar{\bar{A}} = 1$

Redundant Laws:-

i, $A + AB = A$ (Absorption law)

proof:-
$$\begin{aligned} L.H.S &= A + AB \\ &= A(1+B) \quad \because [1+B=1] \\ &= A(1) \\ &= A \end{aligned}$$

ii, $A + \bar{A}B = A + B$

proof:
$$\begin{aligned} L.H.S &= A + \bar{A}B \\ &= A + AB + \bar{A}B \quad \because [A+AB=A] \\ &= A + B(A+\bar{A}) \quad \because [A+\bar{A}=1] \\ &= A + B(1) \\ &= A + B \end{aligned}$$

iii, $(A+B)(A+C) = A + BC$

proof:-
$$\begin{aligned} L.H.S &= (A+B)(A+C) \\ &= A \cdot A + A \cdot C + B \cdot A + B \cdot C \end{aligned}$$

$[\because A \cdot A = A,\ B \cdot A = A \cdot B]$

$= A + AC + A \cdot B + B \cdot C$ $[\because A + A \cdot B = A]$

$= A + AC + BC$

$[\because 1 + C = 1]$

$= A(1+C) + BC$

$= A(1) + BC$

$= A + BC.$

Consensus Theorem

i) $AB + \bar{A}C + BC = AB + \bar{A}C$

proof:- $LHS = AB + \bar{A}C + BC$

$= AB + \bar{A}C + BC(A + \bar{A})$

$= AB + \bar{A}C + BCA + BC\bar{A}$

$= AB(1+C) + \bar{A}C(1+B)$

$= AB(1) + \bar{A}C(1)$

$= AB + \bar{A}C$

ii) $(A+B)(\bar{A}+C)(B+C) = (A+B)(\bar{A}+C)$

proof: $LHS = (A+B)(\bar{A}+C)(B+C)$

$= (A\bar{A} + AC + B\bar{A} + BC)(B+C)$

$= (AC + B\bar{A} + BC)(B+C)$

$= ABC + BB \cdot \bar{A} + B \cdot B \cdot C + A \cdot A \cdot C + B\bar{A}C + B \cdot CC$

$= ABC + B\bar{A} + \underline{BC} + AC + B\bar{A}C + \underline{BC}$ $[\because B \cdot B = B,\ C \cdot C = C,\ A \cdot A = A]$

$= AC(1+B) + B\bar{A}(1+C) + BC$

$= AC + B\bar{A} + BC$ $\quad \because [A+A = A)$

$R.H.S = (A+B)(\bar{A}+C)$

$= A\bar{A} + B\bar{A} + AC + BC$ $\quad \because [A\bar{A} = 0]$

$= AC + B\bar{A} + BC$

$\therefore \quad L.H.S = R.H.S$

$\bar{A} + AB = \bar{A} + B$

proof :- $LHS = \bar{A} + AB$

$= \bar{A}(B+\bar{B}) + AB$

$= \bar{A}B + \bar{A}\bar{B} + AB$

$= \bar{A}\bar{B} + B(\bar{A}+A)$

$= \bar{A}\bar{B} + B$

$= \bar{A} + B$

(or)

$L.H.S = \bar{A} + (AB)$ [$\because$ Distributive Law]

$= (\bar{A}+A)(\bar{A}+B)$

$= \bar{A} + B$

$= R.H.S$

[$\because \; A + \bar{A}B = A+B$]

$\bar{A} + A\bar{B} = \bar{A} + \bar{B}$

proof :- $LHS = \bar{A} + A\bar{B}$

$= \bar{A}(B+\bar{B}) + A\bar{B}$

$= \bar{A}B + \bar{A}\bar{B} + A\bar{B}$

$= \bar{A}B + \bar{B}(A+\bar{A})$

$= \bar{A}B + \bar{B}$

$= \bar{A} + \bar{B}$

(or)

$L.H.S = \bar{A} + (A\bar{B})$

~~$= \bar{A}A + \bar{A}\bar{B}$~~

$= (\bar{A}+A)(\bar{A}+\bar{B})$

$= \bar{A} + \bar{B}$

Transposition theorem:

$AB+\bar{A}C=(A+C)(\bar{A}+B)$

Proof:- $R.H.S = (A+C)(\bar{A}+B)$

$$= A\bar{A}+AB+\bar{A}C+BC$$

$$= 0 + AB+\bar{A}C+BC$$

$$= AB+\bar{A}C+BC(A+\bar{A})$$

$$= AB+\bar{A}C+BCA+BC\bar{A}$$

$$= AB(1+C)+\bar{A}C+\bar{A}BC$$

$$= AB+\bar{A}C+\bar{A}BC$$

$$= AB+\bar{A}C(1+B)$$

$$= AB+\bar{A}C$$

Duality:-

It states that in a two valued Boolean algebra, the dual of an algebra, the dual of an algebraic expression can be obtained simply by interchanging OR and AND operators and by replacing 1s by 0s and 0s by 1s.

Given Expression	Dual Expression
1. $\bar{0}=1$	$\bar{1}=0$
2. $0 \cdot 1 = 0$	$1+0=1$
3. $0 \cdot 0 = 0$	$1+1=1$
4. $1 \cdot 1 = 1$	$0+0=0$
5. $A \cdot 0 = 0$	$A+1=1$
6. $A \cdot 1 = A$	$A+0=A$
7. $A \cdot A = A$	$A+A=A$

8. $A \cdot \overline{A} = 0$ $\quad$ $A + \overline{A} = 1$

9. $A \cdot B = B \cdot A$ $\quad$ $A + B = B + A$

10. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$ $\quad$ $A + (B + C) = (A + B) + C$

11. $A \cdot (B + C) = AB + AC$ $\quad$ $A + (BC) = (A + B)(A + C)$

12. $A(A + B) = A$ $\quad$ $A + AB = A$

13. $A(A \cdot B) = A \cdot B$ $\quad$ $A + A + B = A + B$

14. $\overline{AB} = \overline{A} + \overline{B}$ $\quad$ $\overline{A + B} = \overline{A}\,\overline{B}$

15. $(A + B)(\overline{A} + C)(B + C)' = (A + B)(\overline{A} + C)$ $\quad$ $AB + \overline{A}C + BC = AB + \overline{A}C$

16. $A + \overline{B}C = (A + \overline{B})(A + C)$ $\quad$ $A(\overline{B} + C) = A\overline{B} + AC$

17. $(A + C)(\overline{A} + B) = AB + \overline{A}C$ $\quad$ $AC + \overline{A}B = (A + B)(\overline{A} + C)$

18. $(A + B)(C + D) = AC + AD + BC + BD$ $\quad$ $(AB + CD) = (A + C)(A + D)(B + C)(B + D)$

19. $A + B = AB + \overline{A}B + A\overline{B}$ $\quad$ $AB = (A + B)(\overline{A} + B)(A + \overline{B})$

20. $\overline{A\overline{B} + \overline{A} + AB} = 0$ $\quad$ $\overline{\overline{A + B} \cdot \overline{A} \cdot (A + B)} = 1$

DeMorgans Theorem:

Theorem 1: $\overline{AB} = \overline{A} + \overline{B}$

This theorem states that the complement of a product is equal to addition of the complements.

S.no	A	B	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
1	0	0	1	1	1	1
2	0	1	1	1	0	1
3	1	0	1	0	1	1
4	1	1	0	0	0	0

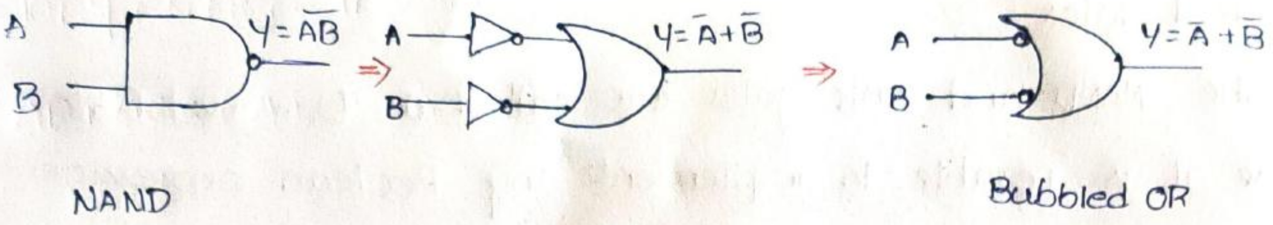


fig:- DeMorgan's First Theorem

Theorem 2: $\overline{A+B} = \bar{A}\cdot\bar{B}$

This theorem states that the complement of a sum is equal to the product of complements.

S.no	A	B	$\overline{A+B}$	$\bar{A}$	$\bar{B}$	$\bar{A}\cdot\bar{B}$
1	0	0	1	1~~0~~	1	1
2	0	1	0	1~~0~~	0	0
3	1	0	0	0	1	0
4	1	1	0	0	0	0

Truth table to verify DeMorgan's Second theorem

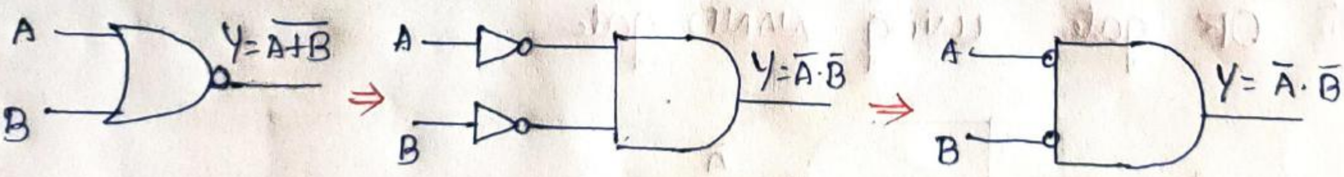


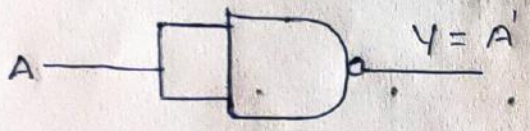
fig:- DeMorgan's Second Theorem

Universal Gates

The NAND and NOR gates are called as Universal Gates because it is possible to implement any Boolean expression with the help of only NAND or only NOR gates.

Realization of gates using NAND gate

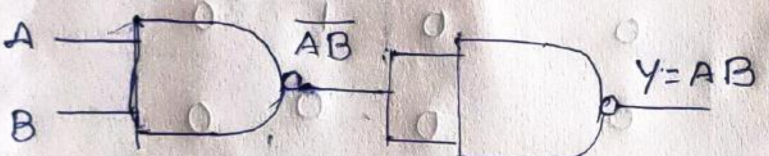
i, NOT gate using NAND gate



A	Y
0	1
1	0

NOT gate

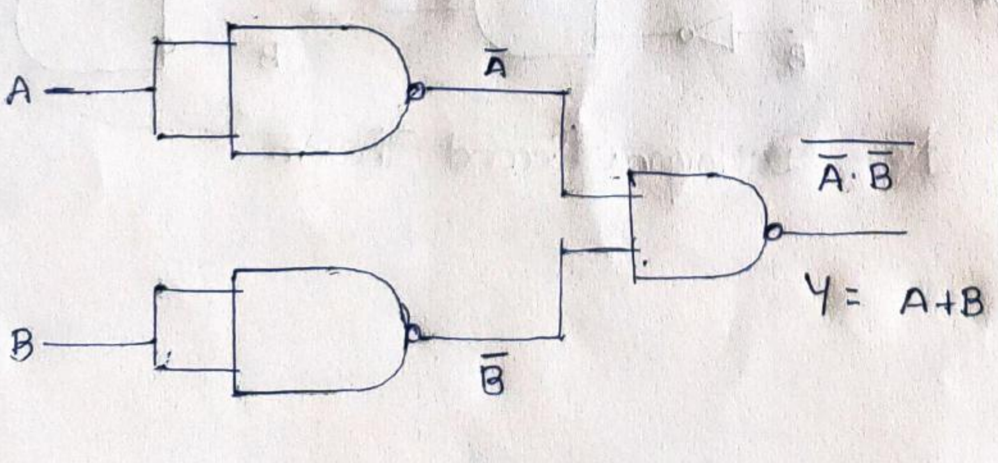
ii, AND gate using NAND gate



A	B	Y=AB
0	0	0
0	1	0
1	0	0
1	1	1

AND gate

iii, OR gate using NAND gate

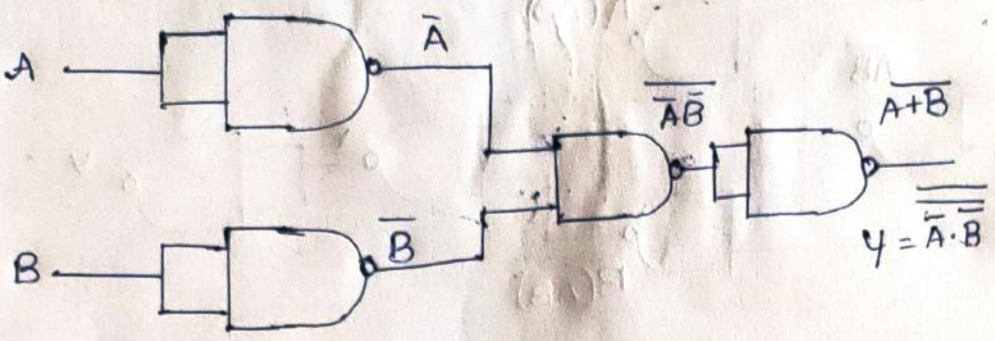


Y = A+B

A	B	Y= A+B
0	0	0
0	1	1
1	0	1
1	1	1

OR gate

(iv) NOR gate using NAND gate

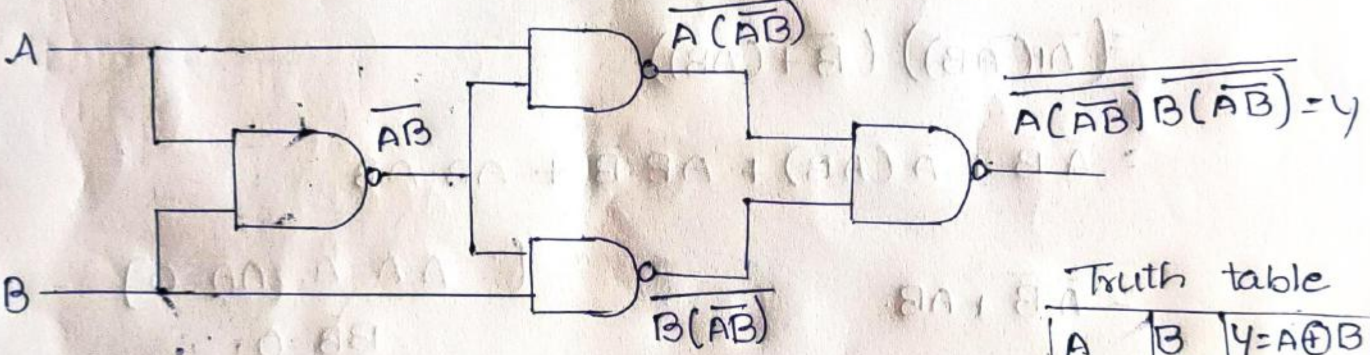

A
B
$\bar{A}$
$\bar{B}$
$\overline{\bar{A}\bar{B}}$
$\overline{A+B}$
$Y = \overline{\overline{\bar{A}\cdot\bar{B}}}$

NOR gate

Truth table

A	$\bar{B}$	$Y = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

(v) EX-OR gate using NAND gate


A
B
$\overline{A(\overline{AB})}$
$\overline{AB}$
$\overline{B(\overline{AB})}$
$\overline{\overline{A(\overline{AB})}\,\overline{B(\overline{AB})}} = Y$

Truth table

A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$Y = \overline{\overline{A(\overline{AB})}\ \overline{B(\overline{AB})}}$

$= A(\overline{AB}) + B(\overline{AB})$ (∵ DeMorgan's theorem

$\bar{A}\cdot\bar{B} = \overline{A+B}$

$\overline{AB} = \bar{A}+\bar{B}$)

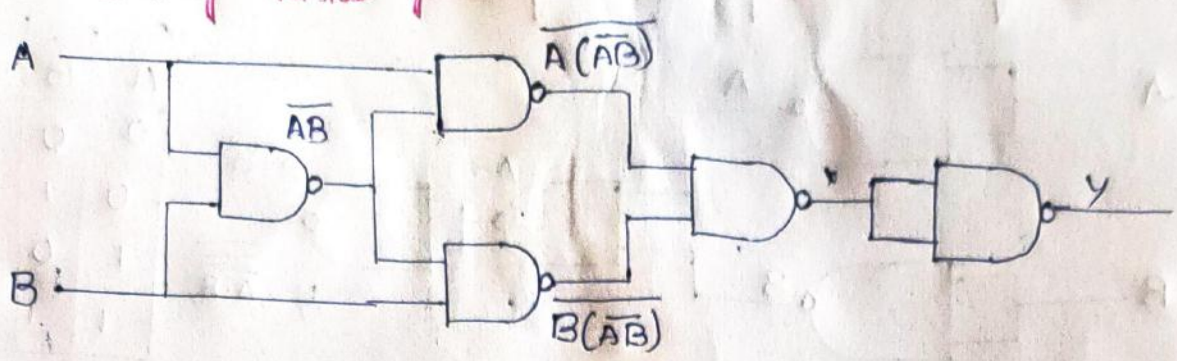
$= A(\bar{A}+\bar{B}) + B(\bar{A}+\bar{B})$

$= A\bar{A} + A\bar{B} + B\bar{A} + B\bar{B}$

(∵ $A\bar{A}=0$, $B\bar{B}=0$)

$= A\bar{B} + \bar{A}B$

vi, X-NOR using NAND gate



Truth table

A	B	$Y = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

$$x = \overline{(\overline{A(\overline{AB})})(\overline{B(\overline{AB})})}$$

$$Y = \overline{\overline{(\overline{A(\overline{AB})})(\overline{B(\overline{AB})})}} \quad (\because \bar{\bar{A}} = A)$$

$$Y = \overline{A(\overline{AB})}\ \overline{B(\overline{AB})}$$

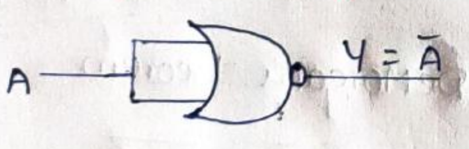
$$= (\bar{A} + (AB))\,(\bar{B} + (AB))$$

$$= \bar{A}\bar{B} + \bar{A}(AB) + AB\cdot\bar{B} + AB\cdot AB$$

$$= \bar{A}\bar{B} + AB. \qquad [\because A\cdot A = A,\ A\bar{A} = 0]$$
$$B\bar{B} = 0$$

Realization of gates using only NOR gate

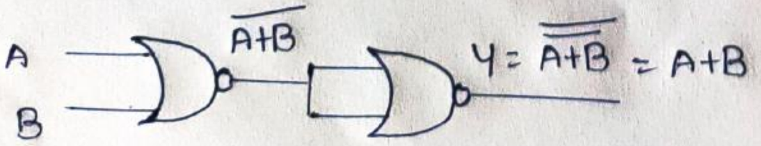
i, NOT gate using NOR gate



Truth table

A	$Y = \bar{A}$
0	1
1	0

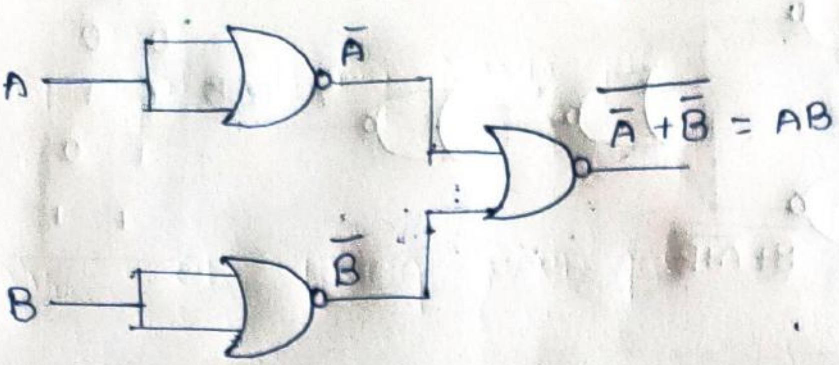
ii, OR gate using NOR gate



Truth table

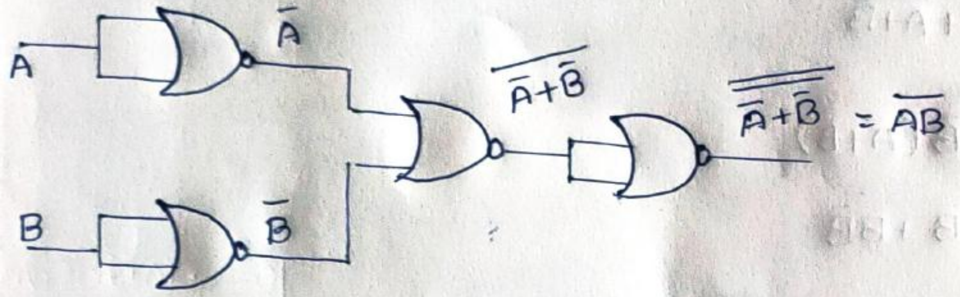
A	B	$Y = A+B$
0	0	0
0	1	1
1	0	1
1	1	1

iii, AND gate using NOR gate



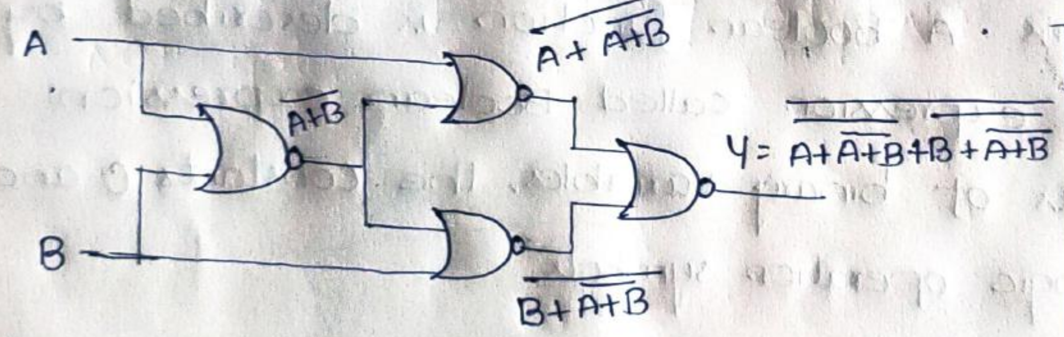
A	B	Y = AB
0	0	0
0	1	0
1	0	0
1	1	1

iv, NAND gate using. NOR gate


A
$\bar{A}$
B
$\bar{B}$
$\overline{\bar{A}+\bar{B}}$
$\overline{\overline{\bar{A}+\bar{B}}} = \overline{AB}$

A	B	$Y=\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

v, X-NOR gate using NOR gate


A
B
$\overline{A+B}$
$\overline{A+\overline{A+B}}$
$\overline{B+\overline{A+B}}$
$Y = \overline{\overline{A+\overline{A+B}}+\overline{B+\overline{A+B}}}$

A	B	$Y=\overline{A \oplus B}$
0	0	~~0~~1
0	1	0
1	0	0
1	1	~~0~~1

$$y = \overline{\left(\overline{A+\overline{A+B}}\right)+\left(\overline{B+\overline{A+B}}\right)}$$

$$= \left(A+\overline{A+B}\right)\cdot\left(B+\overline{A+B}\right)$$

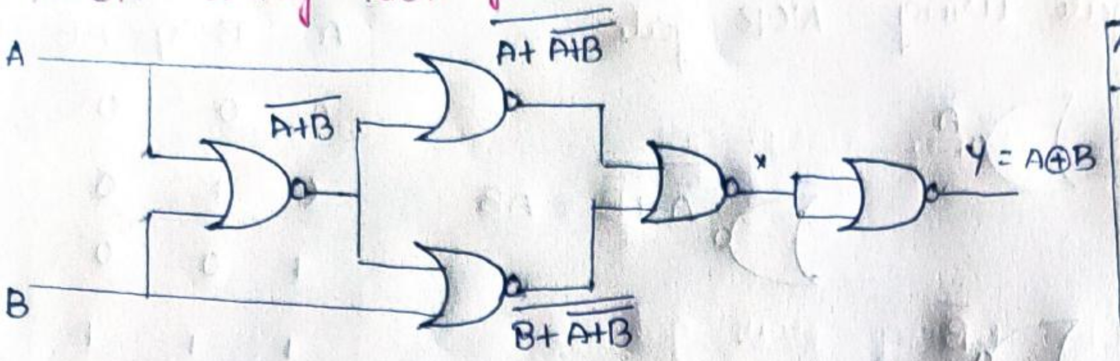
$$= \left(A+(\bar{A}\cdot\bar{B})\right)\left(B+(\bar{A}\cdot\bar{B})\right)$$

$$= (A+\bar{A})(A+\bar{B})(B+\bar{A})(B+\bar{B})$$

$$= (A+\bar{B})(B+\bar{A}) = AB + A\bar{A} + B\bar{B} + \bar{A}\bar{B} = \bar{A}\bar{B} + AB$$

($\because$ $\overline{A+B} = \bar{A}.\bar{B}$, $A+\bar{A}=1$, $B+\bar{B}=1$, $A\bar{A}=0$, $B\bar{B}=0$)

vi, X-OR using NOR gate:



A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

$$x = \overline{\overline{A+\overline{A+B}} + \overline{B+\overline{A+B}}}$$

$$\left[\because \overline{\overline{A}} = A,\ \overline{A+B} = \bar{A}\cdot\bar{B},\ A\bar{A}=0,\ B\bar{B}=0\right]$$

$$Y = \overline{\overline{\overline{\overline{A+\overline{A+B}} + \overline{B+\overline{A+B}}}}}$$

$$= \overline{A+\overline{A+B}} + \overline{B+\overline{A+B}}$$

$$= \bar{A}(A+B) + \bar{B}(A+B)$$

$$= A\bar{A} + \bar{A}B + A\bar{B} + B\bar{B}$$

$$Y = \bar{A}B + A\bar{B}$$

Boolean function and Expression:

Boolean algebra deals with binary variables and logic operations. A Boolean function is described by an algebraic expression called Boolean expression which consists of binary variables, the constants 0 and 1 and the logic operation symbols.

Ex:- $F(A,B,C,D) = A + B\bar{C} + ADC$

Various ways to represent a given function

- Sum of Products form
- Product of Sums form

where Σm represents the sum of all

Sum of Product (Sop) Form:-

- This is also called disjunctive canonical form (DCF) or Expanded Sum of products Form or Canonical sum of products form.
- In this form, the function is the sum of a number of products terms where each product term contains all variables of the function either in complemented or uncomplemented form.
- This can also derived from the truth table by finding the sum of all the terms that corresponds to those combinations for which 'f' assumes the value 1.

Ex:- $f(A,B,C) = \bar{A}B + \bar{B}C$

$$= \bar{A}B(C+\bar{C}) + \bar{B}C(A+\bar{A})$$

$$= \bar{A}BC + \bar{A}B\bar{C} + A\bar{B}C + \bar{A}\bar{B}C$$

- The product term which contains all the variables of the functions either in complemented or uncomplemented form is called a minterm.
- The minterm is denoted as m_0, m_1, m_2 - - -
- Another way of representing the function in canonical Sop form is the showing the sum of minterms for which the function equals to 1.

for ex:- $f(A,B,C) = m_1 + m_2 + m_3 + m_6$

(or)

$$f(A,B,C) = \Sigma m(1,2,3,6)$$

Where Σm represents the sum of all the minterms.

Product of Sum (POS) Form:

- This form is also called as Conjunctive Canonical Form or Expanded product-of-sum Form or Canonical product of Sums Form.
- This is by considering the combinations for which $f=0$
- Each term is a sum of all the variables.
- The function $f(A,B,C) = (\bar{A}+\bar{B})(A+B)$

$$= (\bar{A}+\bar{B}+C\bar{C})(A+B+C\bar{C})$$

$$= (\bar{A}+\bar{B}+C)(\bar{A}+\bar{B}+\bar{C})(A+B+C)(A+B+\bar{C})$$

- The sum term which contains each of the 'n' variables in either complemented or uncomplemented form is called a maxterm.
- Maxterm is represented as $M_0, M_1, M_2 \dots$

Ex:- $f(A,B,C) = M_0 \cdot M_1 \cdot M_7$

or

$f(A,B,C) = \prod M(0,1,7)$

Sop standard form

- The Complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function.

Ex:- $f(A,B,C) = \Sigma m(0,2,6,7)$

$\overline{f(A,B,C)} = \Sigma m(1,3,4,5) = m_1+m_3+m_4+m_5$.

$f = \overline{m_1+m_3+m_4+m_5} = \overline{m_1}\cdot\overline{m_3}\cdot\overline{m_4}\cdot\overline{m_5}$

$= M_1 \; M_3 \; M_4 \; M_5$

$= \pi M \; (1,3,4,5)$

1. Expand $A(\bar{A}+B)(\bar{A}+B+\bar{C})$ to maxterms and minterms

Given pos form is

$$A(\bar{A}+B)(\bar{A}+B+\bar{C})$$

$A = A + (B\bar{B}) + C\bar{C}$

$= (A+B)(A+\bar{B}) + C\bar{C}$

$= (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(A+\bar{B}+\bar{C})$

$\bar{A}+B = \bar{A}+B+C\bar{C}$

$= (\bar{A}+B+C)(\bar{A}+B+\bar{C})$

$\therefore \; A(\bar{A}+B)(\bar{A}+B+\bar{C})$

$= (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+B+\bar{C})$
$(\bar{A}+B+\bar{C})$

$(\because A\cdot A = A)$

$= (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+B+\bar{C})$

$= (000)(001)(010)(011)(100)(101)$

$= M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5$

$= \pi M \; (0,1,2,3,4,5)$

The maxterms M_6, M_7 are missing in the pos form

So, the SOP form will contain the minterms 6 and 7

$= \Sigma m(6,7)$.

2. Convert the expression into their standard SOP form.

$Y = AB + AC + BC$

Given that $Y = AB + AC + BC$

$= AB(C+\bar{C}) + AC(B+\bar{B}) + BC(A+\bar{A})$

$= ABC + AB\bar{C} + ABC + A\bar{B}C + ABC + \bar{A}BC$

$= ABC + AB\bar{C} + A\bar{B}C + \bar{A}BC$

3. Expand $A + BC + ABC$ to maxterms and minterms

Given that $Y = A + BC + ABC$

$= A(B+\bar{B})(C+\bar{C}) + BC(A+\bar{A}) + ABC$

$= (AB+A\bar{B})(C+\bar{C}) + BCA + BC\bar{A} + ABC$

$= \underline{ABC} + A\bar{B}C + AB\bar{C} + A\bar{B}\bar{C} + \underline{ABC} + \bar{A}BC + \underline{ABC}$

$= ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + \bar{A}BC$.

$=$ (000) (00

$= (111) + (110) + (101) + (100) + (011)$

$= m_7, m_6, m_5, m_4, m_3$

$= \Sigma m(3,4,5,6,7)$

Missingminterms are m_0, m_1, m_2

So, the Maxterms are M_0, M_1, M_2

$= \Pi M(0,1,2)$

3 For the given truth table, Write the logic expression in the standard sopform.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Sol:-

A	B	C	Y	
0	0	0	0	
0	0	1	1	→ $\bar{A}\bar{B}C$ (m_1)
0	1	0	0	
0	1	1	0	
1	0	0	1	→ $A\bar{B}\bar{C}$ (m_4)
1	0	1	0	
1	1	0	0	
1	1	1	1	→ ABC (m_7)

} minterms

($\because$ $A=1$, $\bar{A}=0$ for minterms)

$$Y = \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$$

$$= \Sigma m(1, 4, 7)$$

5. For the given truth table Write the logical expression in the standard pos form.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Sol:- Given truth table

A	B	C	Y	
0	0	0	0	$\longrightarrow A+B+C$
0	0	1	1	
0	1	0	0	$\longrightarrow A+\overline{B}+C$
0	1	1	0	$\longrightarrow A+\overline{B}+\overline{C}$
1	0	0	1	
1	0	1	0	$\longrightarrow \overline{A}+B+\overline{C}$
1	1	0	0	$\longrightarrow \overline{A}+\overline{B}+C$
1	1	1	1	

[∵ $A = 0$
$\overline{A} = 1$
for maxterms]

Standard pos form:

$$Y = (A+B+C)(A+\overline{B}+C)(A+\overline{B}+\overline{C})(\overline{A}+B+\overline{C})(\overline{A}+\overline{B}+C)$$

1. Simplify the expression given below

$$Y = AB + (A+B)(\bar{A}+B)$$

Given expression in SOP form

$$Y = AB + (A+B)(\bar{A}+B)$$

$$= AB + (A\bar{A} + AB + \bar{A}B + BB)$$

$$\left[\because AB + AB = AB,\quad A\bar{A} = 0,\quad B \cdot B = B\right]$$

$$Y = AB + \bar{A}B + B$$

$$= B[A + \bar{A}] + B$$

$$= B + B \qquad [\because B+B = B]$$

$$= B$$

2. Simplify the following three variable Boolean expression.

$$Y = \Sigma m(2,4,6)$$

Given expression in SOP form

$$Y = m_2 + m_4 + m_6$$

$$\left[\because m_2 = 010 \quad \bar{A}B\bar{C} \qquad m_4 = 100 \quad A\bar{B}\bar{C} \qquad m_6 = 110 \quad AB\bar{C}\right)$$

$$= \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$$

$$= \bar{A}B\bar{C} + A\bar{C}(\bar{B}+B)$$

$$= \bar{A}B\bar{C} + A\bar{C}$$

$$= \bar{C}(\bar{A}\bar{B} + A)$$

$$\because A + BC = (A+B)(A+C)$$

$$= \bar{C}(A+\bar{A})(A+\bar{B}) \quad = \bar{C}(A+\bar{B})$$

Simplify the following three variable logic expression

$$Y = \prod M(1,3,5)$$

Given expression in pos form

$Y = M1 \cdot M3 \cdot M5$

$(\because M1 = 001$
$A+B+\bar{C}$
$M3 = 011$
$A+\bar{B}+\bar{C}$
$M5 = 101$
$\bar{A}+B+\bar{C})$

$= (A+B+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+B+\bar{C})$

$= (A+B+\bar{C})[A\bar{A}+AB+A\bar{C}+\bar{B}\bar{A}+B\bar{B}$
$+\bar{B}\bar{C}+\bar{A}\bar{C}+B\bar{C}+\bar{C}\bar{C}]$

$(\because A\bar{A}=0,\ \bar{C}\bar{C}=\bar{C},\ B\bar{B}=0)$

$= (A+B+\bar{C})(AB+A\bar{C}+\bar{A}\bar{B}+\bar{B}\bar{C}+\bar{A}\bar{C}+B\bar{C}+\bar{C})$

$= (A+B+\bar{C})(AB+\bar{C}(A+\bar{A})+\bar{A}\bar{B}+\bar{C}(B+\bar{B})+\bar{C})$

$(\because A+\bar{A}=1,\ B+\bar{B}=1)$

$= (A+B+\bar{C})(AB+\bar{C}+\bar{A}\bar{B}+\bar{C}+\bar{C})$

$= (A+B+\bar{C})(AB+\bar{A}\bar{B}+\bar{C})$

$= A\cdot AB + A\bar{A}\bar{B} + A\bar{C} + A\cdot B\cdot B + \bar{A}B\bar{B} + B\bar{C} + AB\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{C}\bar{C}$

$= AB + 0 + A\bar{C} + A\cdot B + 0 + B\bar{C} + AB\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{C}$

$= AB + A\bar{C} + B\bar{C} + AB\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{C}$

$= AB(1+\bar{C}) + A\bar{C} + B\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{C}$

$= AB + A\bar{C} + B\bar{C} + \bar{A}\bar{B}\bar{C} + \bar{C}$

$= AB + \bar{C}\,(A + B + \bar{A}\bar{B} + 1)$

$y = AB + \bar{C}$ $\quad\quad$ $(\because A + B + \bar{A}\bar{B} + 1 = 1)$

Karanaugh Maps (or) K-map:

The K-map is a chart or a graph, composed of an arrangement of adjacent cells, each representing a particular combination of variables in sum or product form.

- The k-map is systematic method of simplifying the Boolean expression.

Two Variable k-map:-

A two variable expression can have $2^2 = 4$ possible combinations of the input variables A and B.

SOP Form:-

* The 2variable K-map has $2^2 = 4$ squares. These squares are called cells.

* A '1' is placed in any square indicates that corresponding minterm is included in the output expression, and a 0 or no entry in any square indicates that the corresponding minterm does not appear in the expression for output.

A \ B	0	1
0	$\bar{A}\bar{B}$	$\bar{A}B$
1	$A\bar{B}$	AB

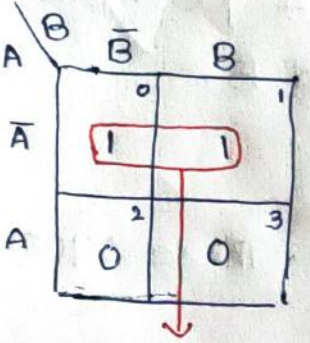
POS form:-

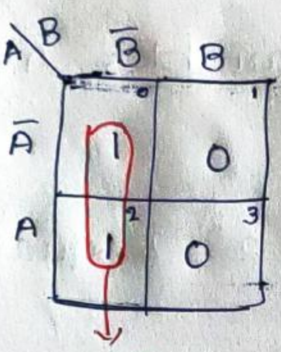
Each sum term in the standard POS expression is called a Maxterm. A function in two variables (A, B) has 4 possible maxterms, $A+B$, $A+\bar{B}$, $\bar{A}+B$ and $\bar{A}+\bar{B}$. They are represented as M_0, M_1, M_2 and M_3 respectively.

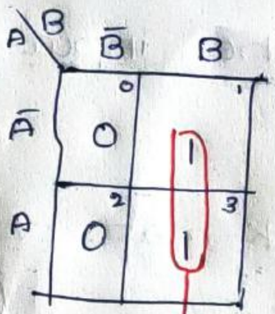
A \ B	0	1
0	$A+B$ (0)	$A+\bar{B}$ (1)
1	$\bar{A}+B$ (2)	$\bar{A}+\bar{B}$ (3)

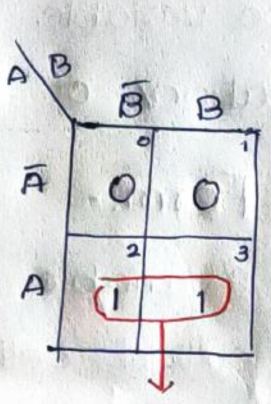
SOP Form

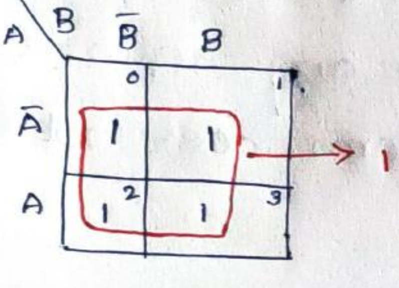
The possible minterm grouping in a two variable k-map are shown below


$f = \bar{A}$


$f = \bar{B}$


$f = B$


$f = A$

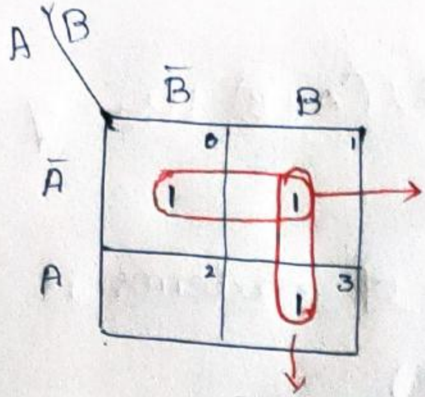

$f = 1$

Reduce the expression $f = \bar{A}\bar{B} + \bar{A}B + AB$ using K-map

Given expression $f = \bar{A}\bar{B} + \bar{A}B + AB$ expressed in minterms

$$f = m_0 + m_1 + m_3$$

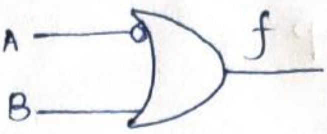
$$f = \Sigma m(0,1,3)$$



$\bar{A}\bar{B} + \bar{A}B$

$\bar{A}(\bar{B}+B)$

$\bar{A}$



$\bar{A}B + AB$

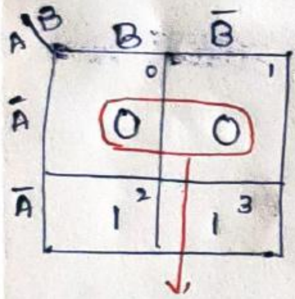
$B(\bar{A}+A)$

B

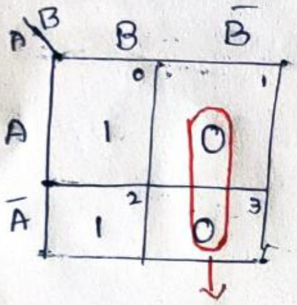
$f = \bar{A} + B$

POS form

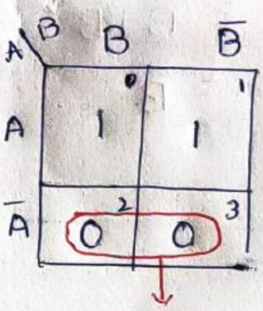
The possible ~~min~~ maxterm grouping in a two variable k-map are shown below.



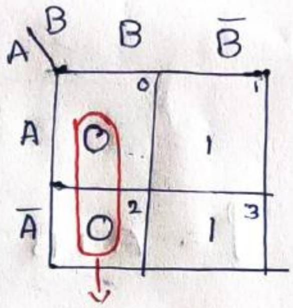
$f = \bar{A}$



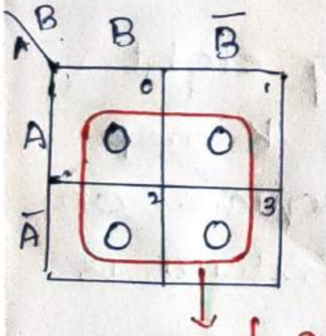
$f = \bar{B}$



$f = \bar{A}$



$f = B$

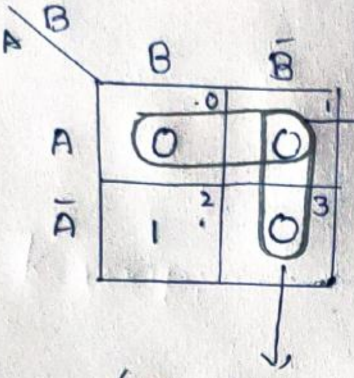


$f = 0$

Reduce the expression $f=(A+\bar{B})(\bar{A}+\bar{B})(A+B)$ using K-map

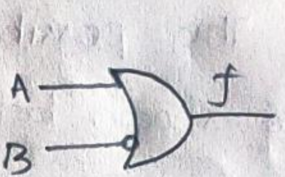
sol:- The given expression in terms of maxterms is

$$f = \Pi M(0,1,3)$$



$$(A+B)(A+\bar{B})$$
$$= AA + A\bar{B} + AB + B\bar{B}$$
$$= A + A(B+\bar{B})$$
$$= A + A$$
$$= A$$

$$(\bar{A}+\bar{B})(A+\bar{B})$$
$$= A\bar{A} + \bar{A}\bar{B} + A\bar{B} + \bar{B}\bar{B}$$
$$= \bar{B}(\bar{A}+A) + \bar{B}$$
$$= \bar{B} + \bar{B}$$
$$= \bar{B}$$



$$f = A + \bar{B}$$

Three Variable k-map

A function in three variables (A,B,C) can be expressed in SOP and POS form having eight possible combination. A three variable K-map have 8 squares or cells and each square on the map represents a minterm or maxterm is shown in the figure below.

A \ BC	00	01	11	10
0	0 $\bar{A}\bar{B}\bar{C}$ (m_0)	1 $\bar{A}\bar{B}C$ (m_1)	3 $\bar{A}BC$ (m_3)	2 $\bar{A}B\bar{C}$ (m_2)
1	4 $A\bar{B}\bar{C}$ (m_4)	5 $A\bar{B}C$ (m_5)	7 ABC (m_7)	6 $AB\bar{C}$ (m_6)

minterms

A \ BC	00	01	11	10
0	0 $A+B+C$ (M_0)	1 $A+B+\bar{C}$ (M_1)	3 $A+\bar{B}+\bar{C}$ (M_3)	2 $A+\bar{B}+C$ (M_2)
1	4 $\bar{A}+B+C$ (M_4)	5 $\bar{A}+B+\bar{C}$ (M_5)	7 $\bar{A}+\bar{B}+\bar{C}$ (M_7)	6 $\bar{A}+\bar{B}+C$ (M_6)

maxterms

Four Variable k-map

A four variable (A,B,C,D) expression can have $2^4 = 16$ possible combinations of input variables. A four variable k-map has $2^4 = 16$ squares or cells and each square on the map represents either a minterm or a maxterm as shown in figure below.

AB \ CD	00	01	11	10
00	0 $\bar{A}\bar{B}\bar{C}\bar{D}$ (m_0)	1 $\bar{A}\bar{B}\bar{C}D$ (m_1)	3 $\bar{A}\bar{B}CD$ (m_3)	2 $\bar{A}\bar{B}C\bar{D}$ (m_2)
01	4 $\bar{A}B\bar{C}\bar{D}$ (m_4)	5 $\bar{A}B\bar{C}D$ (m_5)	7 $\bar{A}BCD$ (m_7)	6 $\bar{A}BC\bar{D}$ (m_6)
11	12 $AB\bar{C}\bar{D}$ (m_{12})	13 $AB\bar{C}D$ (m_{13})	15 $ABCD$ (m_{15})	14 $ABC\bar{D}$ (m_{14})
10	8 $A\bar{B}\bar{C}\bar{D}$ (m_8)	9 $A\bar{B}\bar{C}D$ (m_9)	11 $A\bar{B}CD$ (m_{11})	10 $A\bar{B}C\bar{D}$ (m_{10})

Sop form (minterms)

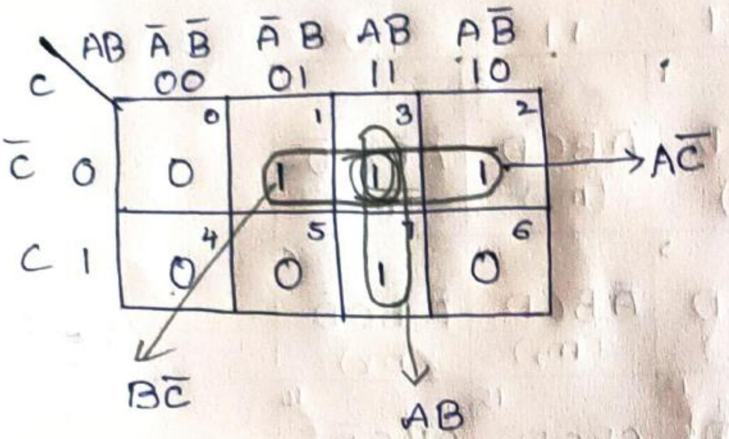
AB \ CD	00	01	11	10
00	0 $A+B+C+D$ (M_0)	1 $A+B+C+\bar{D}$ (M_1)	3 $A+B+\bar{C}+\bar{D}$ (M_3)	2 $A+B+\bar{C}+D$ (M_2)
01	4 $A+\bar{B}+C+D$ (M_4)	5 $A+\bar{B}+C+\bar{D}$ (M_5)	7 $A+\bar{B}+\bar{C}+\bar{D}$ (M_7)	6 $A+\bar{B}+\bar{C}+D$ (M_6)
11	12 $\bar{A}+\bar{B}+C+D$ (M_{12})	13 $\bar{A}+\bar{B}+C+\bar{D}$ (M_{13})	15 $\bar{A}+\bar{B}+\bar{C}+\bar{D}$ (M_{15})	14 $\bar{A}+\bar{B}+\bar{C}+D$ (M_{14})
10	8 $\bar{A}+B+C+D$ (M_8)	9 $\bar{A}+B+C+\bar{D}$ (M_9)	11 $\bar{A}+B+\bar{C}+\bar{D}$ (M_{11})	10 $\bar{A}+B+\bar{C}+D$ (M_{10})

pos form (maxterms)

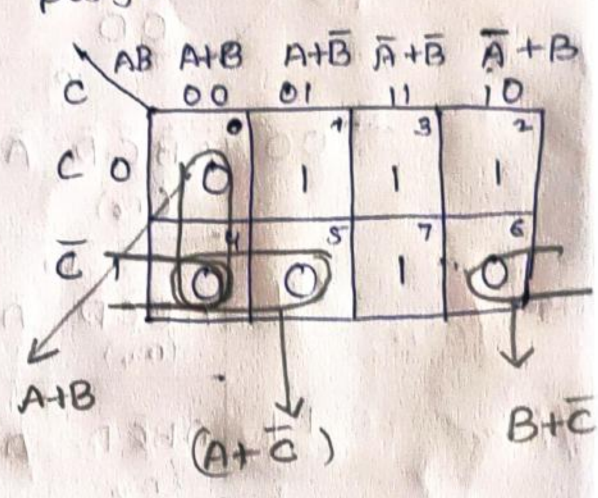
- For the k-map shown in figure. Write the simplified Boolean expression in SOP form and pos form

C \ AB	00	01	11	10
0	0	1	1	1
1	0	0	1	0

SOP form



pos form



$Y = AB + B\bar{C} + A\bar{C}$

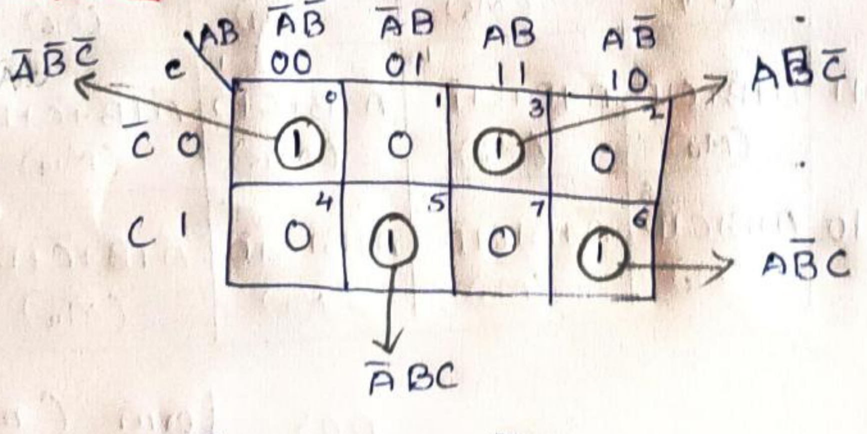
$Y = (A+B)(B+\bar{C})(A+\bar{C})$

- Write down the simplified Boolean expression for the k-map shown in figure.

C \ AB	00	01	11	10
0	1	0	1	0
1	0	1	0	1

sol:-

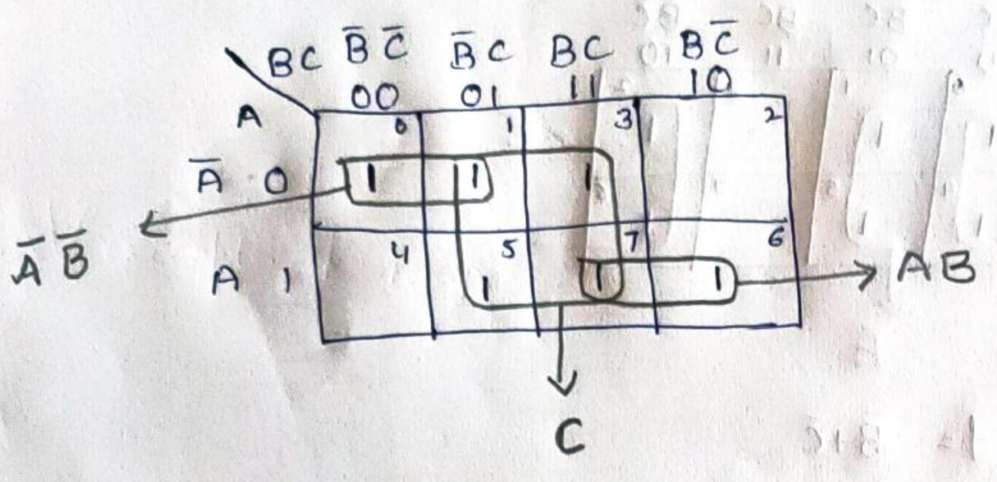
Sop form



$Y = \bar{A}\bar{B}\bar{C} + AB\bar{C} + \bar{A}BC + A\bar{B}C$

Using K-map, realize the following expression

$$f(A,B,C) = \Sigma m(0,1,3,5,6,7)$$

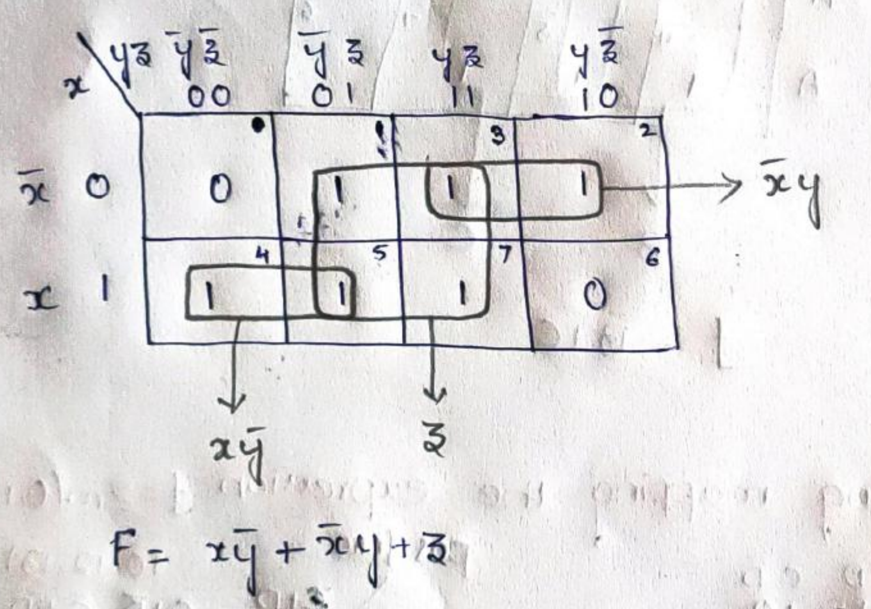


A \ BC	$\bar{B}\bar{C}$ 00	$\bar{B}C$ 01	BC 11	$B\bar{C}$ 10
$\bar{A}$ 0	1	1	1	
A 1		1	1	1

$Y = \bar{A}\bar{B} + AB + C$

Implement the following Boolean function with NAND gates.

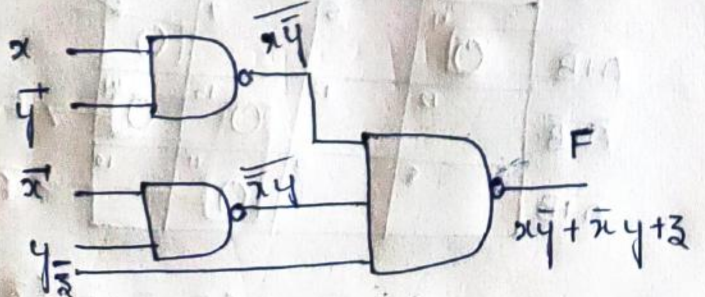
$$F(x,y,z) = \Sigma(1,2,3,4,5,7)$$



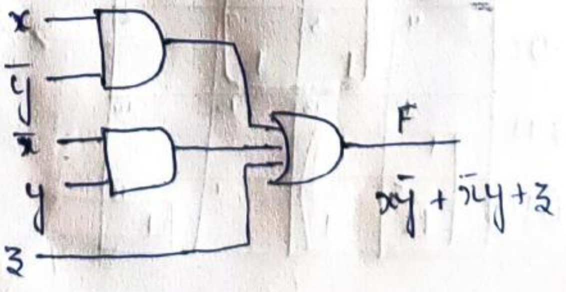
x \ yz	$\bar{y}\bar{z}$ 00	$\bar{y}z$ 01	yz 11	$y\bar{z}$ 10
$\bar{x}$ 0	0	1	1	1
x 1	1	1	1	0

$F = x\bar{y} + \bar{x}y + z$

NAND gate implementation

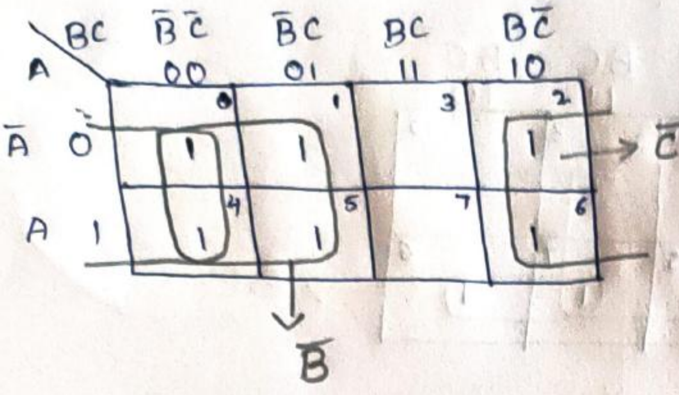


using AND & OR gates



- Simplify the Boolean function using K-maps

$$f = \Sigma m(0, 1, 2, 4, 5, 6)$$

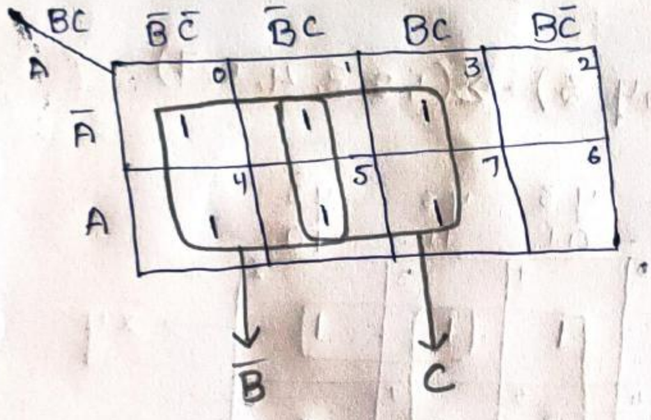


A \ BC	$\bar{B}\bar{C}$ 00	$\bar{B}C$ 01	BC 11	$B\bar{C}$ 10
$\bar{A}$ 0	1	1		1
A 1	1	1		1

$$f = \bar{B} + \bar{C}$$

- Simplify the following expression using K-maps

$$f = \Sigma m(0, 1, 3, 4, 5) 7)$$

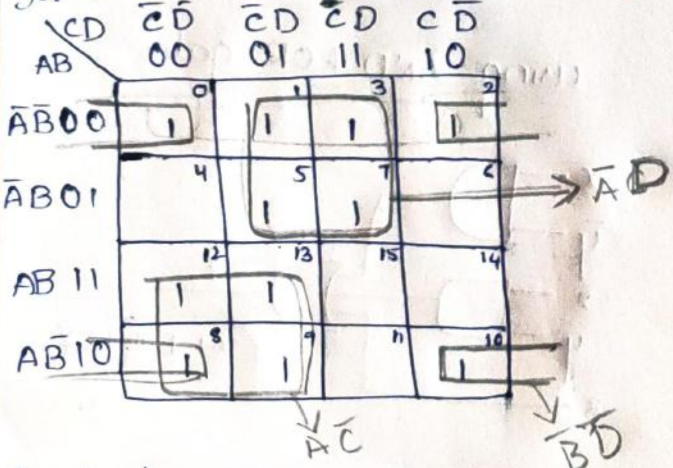


A \ BC	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	1	1	1	
A	1	1	1	

$$f = \bar{B} + C$$

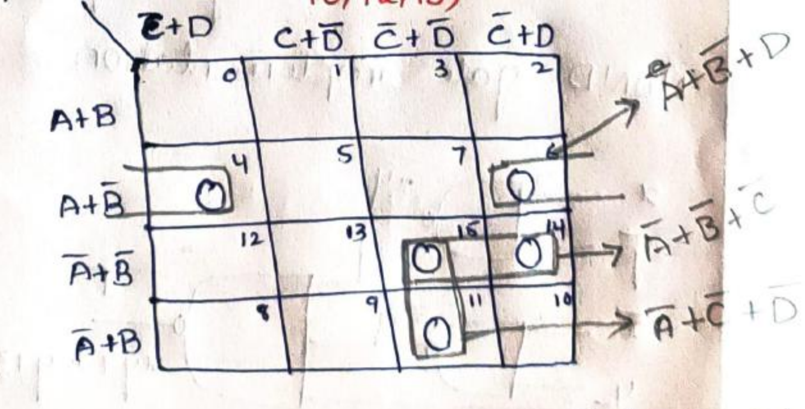
Reduce using mapping the expression $f = \Sigma m(0,1,2,3,5,7,8,9,$ $10,12,13)$

SOP form



AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	1	1	1	1
$\bar{A}B$ 01		1	1	
AB 11	1	1		
$A\bar{B}$ 10	1	1		1

pos form



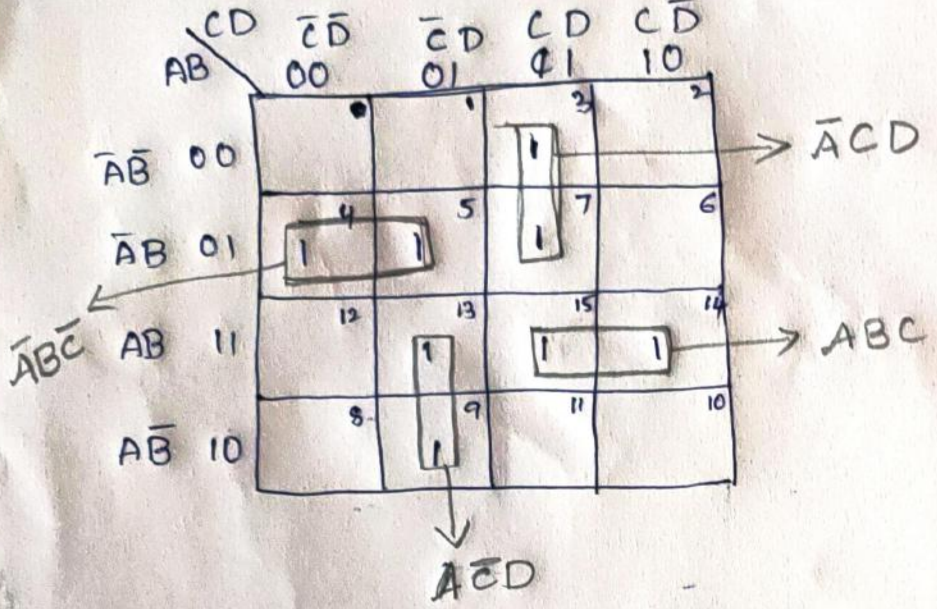
	$C+D$	$C+\bar{D}$	$\bar{C}+\bar{D}$	$\bar{C}+D$
A+B				
$A+\bar{B}$	0			0
$\bar{A}+\bar{B}$			0	0
$\bar{A}+B$			0	

SOP form: $f = \bar{B}\bar{D} + A\bar{C} + \bar{A}D$

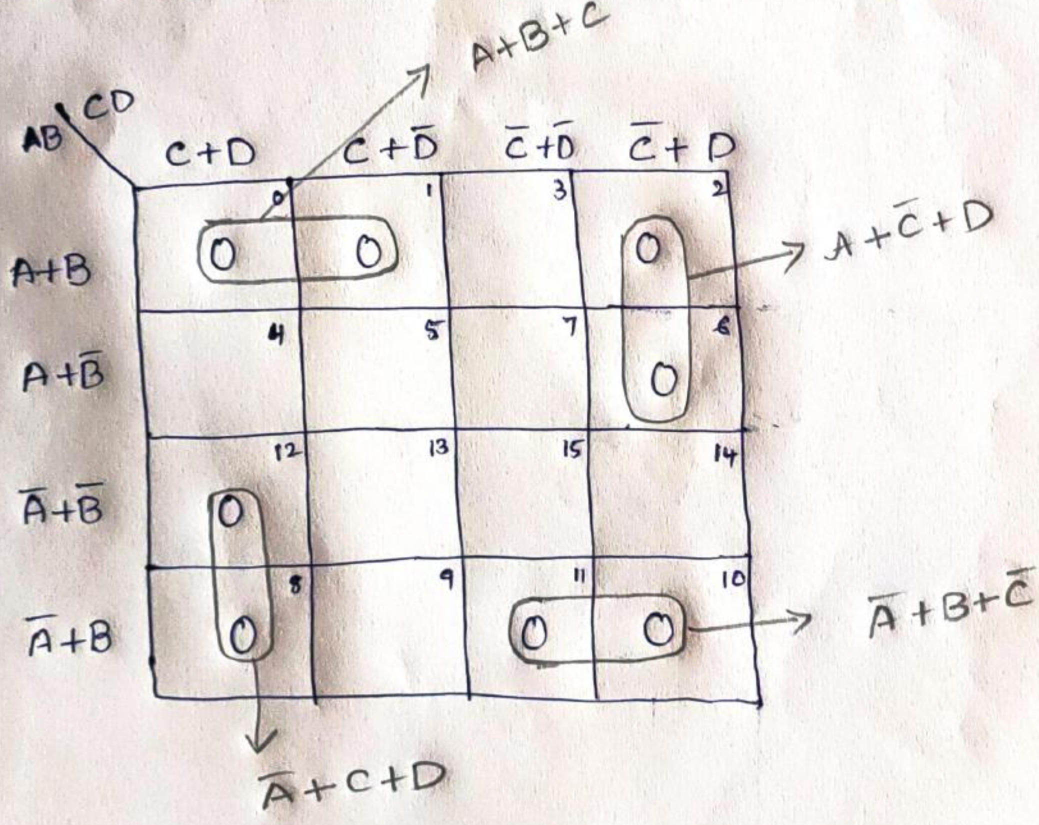
pos form $f = (A+\bar{B}+D)(\bar{A}+\bar{C}+\bar{D})(\bar{A}+\bar{B}+\bar{C})$

Simplify the following boolean function by using four-variable k-map.

$$f = \Sigma(3,4,5,7,9,13,14,15)$$



Sop form $f = A\bar{C}D + ABC + \bar{A}B\bar{C} + \bar{A}CD$



pos form ⇒ $f = (\bar{A}+C+D)(\bar{A}+B+\bar{C})(A+\bar{C}+D)(A+B+C)$

1. Convert the Gray code 1101 to binary

Gray	1	1	0	1
Binary	1	0	0	1

2. 1 0 1 1 0 1

Gray	1	0	1	1	0	1
Binary	1	1	0	1	1	0

Error Detecting and Correcting Codes :

→ Codes which allow only error detection are called "Error Detecting codes".

→ Codes which allow error detection and correction are called "Error Detecting and Correcting codes".

1. Parity Bit

3-bit Message			Message with odd parity		Msg. with even parity	
A	B	C	Message	parity	Message	parity
0	0	0	000	1	000	0
0	0	1	001	0	001	1
0	1	0	010	0	010	1
0	1	1	011	1	011	0
1	0	0	100	0	100	1
1	0	1	101	1	101	0
1	1	0	110	1	110	0
1	1	1	111	0	111	1

Hamming code :

No. of parity bits.

$$2^P \geq x+P+1$$

$P \rightarrow$ No. of parity bits

$x \rightarrow$ No. of information bits.

Example :

- For 4 bit information, let $x=4$

$P \rightarrow$ Trial & error let $P=2$

$2^2 \geq 4+2+1$

$4 \geq 7$ ✗

let $P=3$

$2^3 \geq 4+3+1$

$8 \geq 8$ ✓

∴ Three parity bits are required to provide single error correction for four information bits.

Error-Correcting codes :

1. 7-bit Hamming code

P_1 P_2 D_3 P_4 D_5 D_6 D_7

$P_1 \rightarrow 1, 3, 5, 7$

$P_2 \rightarrow 2, 3, 6, 7$

$P_4 \rightarrow 4, 5, 6, 7$

2) The 12-bit Hamming code

$P_1 \quad P_2 \quad D_3 \quad P_4 \quad D_5 \quad D_6 \quad D_7 \quad P_8 \quad D_9 \quad D_{10} \quad D_{11} \quad D_{12}$

$P_1 \rightarrow 1,3,5,7,9,11$

$P_2 \rightarrow 2,3,6,7,10,11$

$P_4 \rightarrow 4,5,6,7,12$

$P_8 \rightarrow 8,9,10,11,12$

3) The 15-bit Hamming code

$P_1 \quad P_2 \quad D_3 \quad P_4 \quad D_5 \quad P_6 \quad D_7 \quad P_8 \quad D_9 \quad D_{10} \quad D_{11} \quad D_{12} \quad D_{13} \quad D_{14} \quad D_{15}$

$P_1 \rightarrow 1,3,5,7,9,11,13,15$

$P_2 \rightarrow 2,3,6,7,10,11,14,15$

$P_4 \rightarrow 4,5,6,7,12,13,14,15$

$P_8 \rightarrow 8,9,10,11,12,13,14,15$

1. Encode the binary coord $\underbrace{1011}_{x=4}$ into seven bit even parity hamming code.

Step 1: find the no. of parity bits required

let $p=3$

$2^p \geq x+p+1$

$2^3 \geq 4+3+1$

$8 \geq 8$ ✓

Three parity bits are sufficient

Total code bits $= x+p = 4+3 = 7$

Step 2) Construct a bit location table

Bit designation	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	7	6	5	4	3	2	1
Binary location number	111	110	101	100	010	010	001
Information bits	1	0	1		1		
Parity bits				0		0	1

step 3) Determine the parity bits

even

↓ 1 $P_1 \rightarrow 3\ 5\ 7 \rightarrow 1\ 1\ 1$

0 $P_2 \rightarrow 3\ 6\ 7 \rightarrow 1\ 0\ 1$

0 $P_3 \rightarrow 5\ 6\ 7 \rightarrow 1\ 0\ 1$

∴ Seven bit Hamming code is 1 0 1 0 1 0 1

2. Determine the single error-correcting code for the information code 10111 for odd parity.

step 1: Find the number of parity bits required

let $p=3$ $\qquad x=5$

$$2^p \geq x+p+1$$

$$2^3 \geq 5+3+1$$

$$8 \geq 9 \quad \times$$

let $p=4$

$$2^p \geq x+p+1$$

$$2^4 \geq 5+4+1$$

$$16 \geq 10 \ \checkmark$$

Hence four bits are sufficient

Total code bit = 5+4 = 9

Step 2 : Construct a bit location table

Bit designation	D_9	P_8	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	9	8	7	6	5	4	3	2	1
Binary location number	1001	1000	0111	0110	0101	0100	0011	0010	0001
Information bits	1		0	1	1	1	1	1	0
parity bits		0							

odd

Step 3 : Determine the parity Bits

0 For P_1 : 3 5 7 9   1 1 0 1

1 For P_2 : 3 6 7   1 1 0

1 For P_4 : 5 6 7   1 1 0

For P_8 : Bit P_8 checks bit locations 8 and $\overset{1}{9}$. and must be 0 to have an odd parity.

Step 4 : Enter the parity bits into the table to form a nine bit Hamming code

1 0 0 1 1 1 1 1 0

Detecting and Correcting an Error :

1. Assume that the even parity Hamming code in example 0 1 1 0 0 1 is transmitted and that 0 1 0 0 0 1 1 is received. The receiver does not know what was transmitted. Determine bit location where error has occured using received code.

7

Step 1: Construct the bit location table

Bit designation	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	7	6	5	4	3	2	1
Binary location number	111	110	101	100	011	010	001
Received code	0	1	0	0	0	1	1

Step 2 : Check for parity bits even

1 For P_1 : checks locations 1,3,5,7 → 1 0 0 0 (LSB)

0 For P_2 : checks locations 2,3,6,7 → 1 0 1 0

1 For P_4 : checks locations 4,5,6,7 → 0 0 1 0

∴ The resultant word } 1 0 1

This says that the bit in the number 5 locations is in error. It is 0 & it should be 1.

∴ The correct code 0 1 1 0 0 1 1.

he Hamming Code 1011 01101 is received. Correc
it if any errors. There are four parity bits and odd
arity is used.

ALPHANUMERIC CODES

→ Alphanumeric codes are codes used to encode the characters of alphabet in addition to the decimal digits.

The ASCII code: American Standard Code for Information Interchange

→ It is basically a 7-bit code

i.e $2^7 = 128$

Table: The ASCII code.

	MSBs							
SBs	000	001	010	011	100	101	110	111
0000	NUL	DLE	space	0	@	P	P	
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DLE

Abbreviations:

ACK - Acknowledge

BEL - Bell

BS - Backspace

CAN - cancel

CR - carriage return

DC1 - Direct Control 1

DC2 - Direct Control 2

DC3 - " " 3

DC4 - " " 4

DEL - Delete idle

DLE - Data link escape

EM - End of medium

ENQ - Enquiry

EOT - End of transmission

ESC - Escape

ETB - End of transmission block

ETX - End of text

FF - form feed

FS - form Separator

GS - Group Separator

HT - Horizontal Tab

LF - Line feed

NAK - Negative Acknowledge

NUL - Null

RS - Record Separator

SI - Shift in

SO - shift out

SOH - start of heading

STX - Substitute

SYN - Synchronous idle

US - Unit Separator

VT - Vertical Tab

The EBCDIC Code : Extended Binary Coded Decimal Interchange Code.

$\therefore\ 2^8 = 256$

MSD (hex)

LSD (hex)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&							{	}	\	0
1	SOH	DC1	SOS				/		a	j	~		A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	DC3							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	EOB	YC					f	o	w		F	O	W	6
7	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM							i	r	z		I	R	Z	9
A	SMM	CC	SM		¢	!	I	:								
B	VT				.	$	,	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	I	'	?	,								

UNIT-2

Combinational Circuit :-

A Combinational circuit may be defined as a logic circuit the output of which depends only upon the combination of the inputs. The output does not depend on the past value of inputs or outputs. Therefore, combinational circuits do not need any memory.

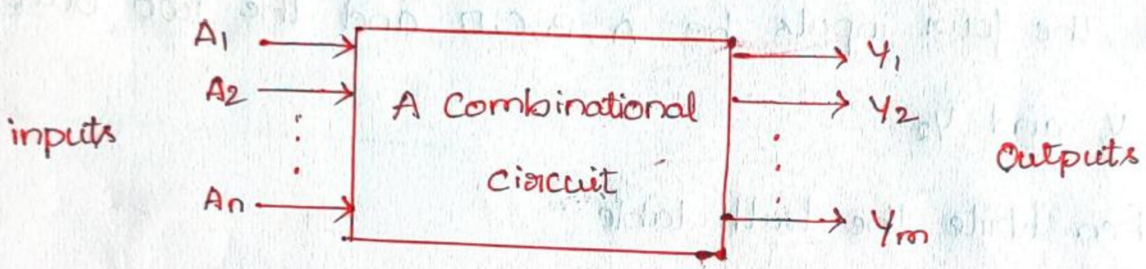


fig:- Block diagram of a Combinational circuit.

A Combinational circuit can have a number of inputs and a number of outputs. The above figure has n inputs and m outputs. Between the inputs and outputs, logic gates are connected, and hence, combinational circuit basically consists of logic gates.

The Various Steps Involved in Designing Procedure of a Combinational logic may be listed Under:

- we will be given a problem
- Then, we determine the number of inputs and outputs and assign letter symbols to input and output variables.
- After that we write a truth table relating the inputs and

outputs.

- Then, Write k-map for each output and obtain the simplified Boolean expression for each output.
- Lastly draw the logic diagram.

Ex:- A circuit has four inputs and two outputs. One of the outputs is high when majority of inputs are high. The second output is high only when all inputs are of same type. Design the combinational circuit.

Sol:- i, Let the four inputs be A, B, C, D and the two outputs be Y_1 and Y_2.

ii, Then Write the truth table

Decimal	Inputs				outputs	
	A	B	C	D	Y_1	Y_2
0	0	0	0	0	0	1
1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	0	0
5	0	1	0	1	0	0
6	0	1	1	0	0	0
7	0	1	1	1	1	0
8	1	0	0	0	0	0
9	1	0	0	1	0	0
10	1	0	1	0	0	0
11	1	0	1	1	1	0
12	1	1	0	0	0	0
13	1	1	0	1	1	0
14	1	1	1	0	1	0
15	1	1	1	1	1	1

iii, Then. Write k-map for each output and get simplified expression.

for output Y_1

$\begin{array}{c}CD\\AB\end{array}$	$\bar C\bar D$ 00	$\bar C D$ 01	CD 11	$C\bar D$ 10
$\bar A\bar B$ 00	^{0} 0	^{1} 0	^{3} 0	^{2} 0
$\bar A B$ 01	^{4} 0	^{5} 0	^{7} 1	^{6} 0
AB 11	^{12} 0	^{13} 1	^{15} 1	^{14} 1
$A\bar B$ 10	^{8} 0	^{9} 0	^{11} 1	^{10} 0

→ BCD

→ ABC

ABD

ACD

for output Y_2

$\begin{array}{c}CD\\AB\end{array}$	$\bar C\bar D$ 00	$\bar C D$ 01	CD 11	$C\bar D$ 10
$\bar A\bar B$ 00	^{0} 1	^{1} 0	^{3} 0	^{2} 0
$\bar A B$ 01	^{4} 0	^{5} 0	^{7} 0	^{6} 0
AB 11	^{12} 0	^{13} 0	^{15} 1	^{14} 0
$A\bar B$ 10	^{8} 0	^{9} 0	^{11} 0	^{10} 0

→ $\bar A\bar B\bar C\bar D$

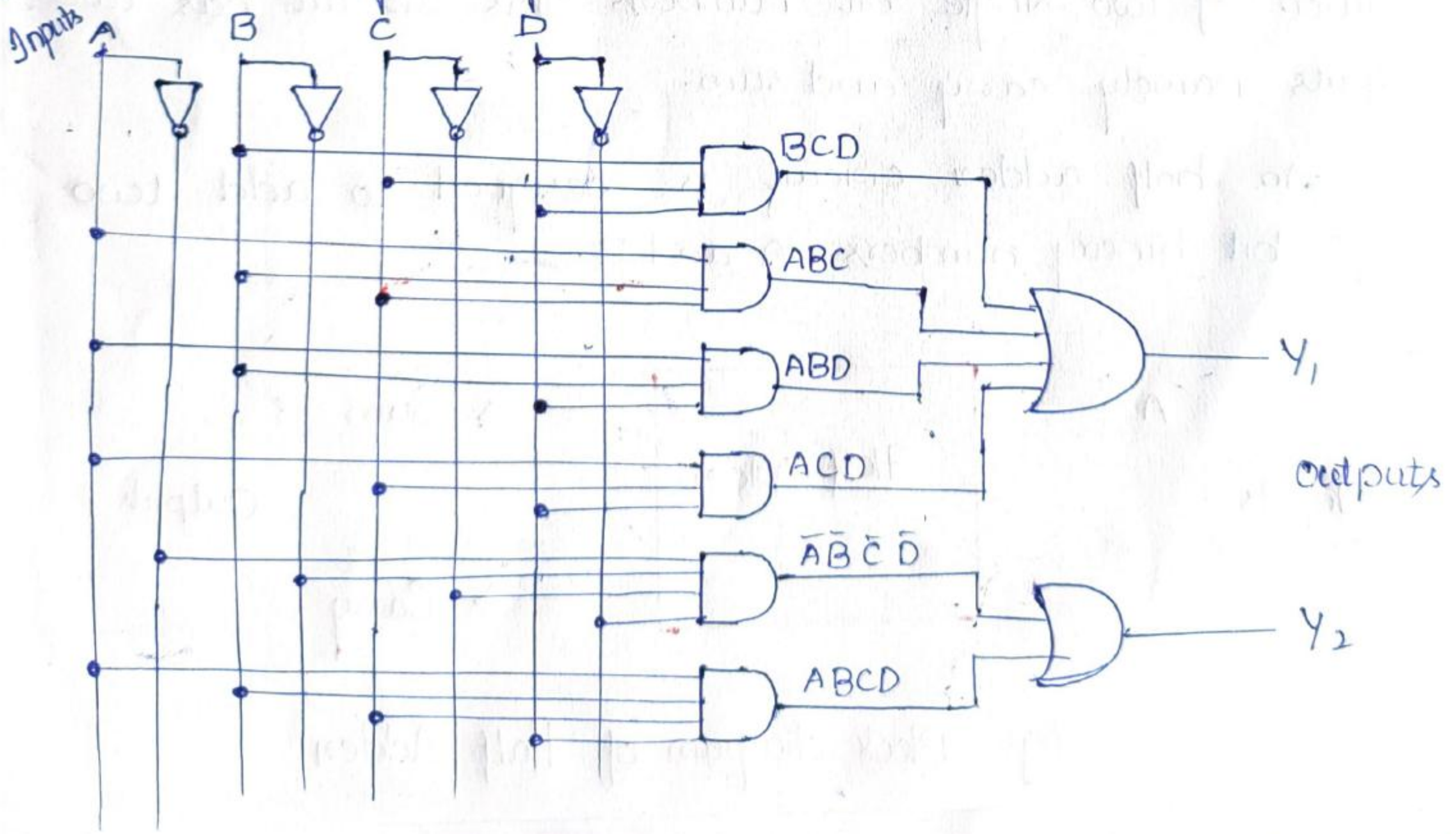
→ ABCD

Simplified expressions

$Y_1 = BCD + ABC + ABD + ACD$

$Y_2 = \bar A\bar B\bar C\bar D + ABCD$

iv, Draw the logic diagram.



Classification of Combinational Circuits

The Combinational circuits may be classified as

i) Code converters

ii) Adders

iii) Subtractors

iv) Comparators

- Adders:

Addition of two binary digits is most basic operation performed by the digital computers.

classification of Binary Adders:

i) Half adder, and

ii) Full adder

Half Adder:-

Half adder is a combinational logic circuit with two inputs and two outputs. It is the basic building block for addition of two single bit numbers. This circuit has two outputs namely carry and sum.

An half adder circuit is designed to add two single bit binary numbers A and B.

(i)

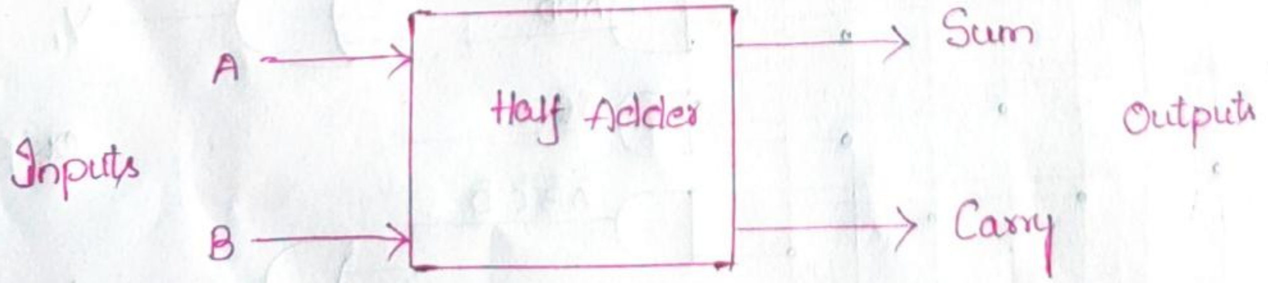
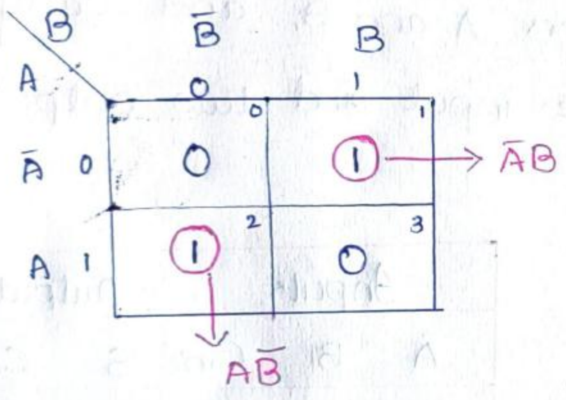


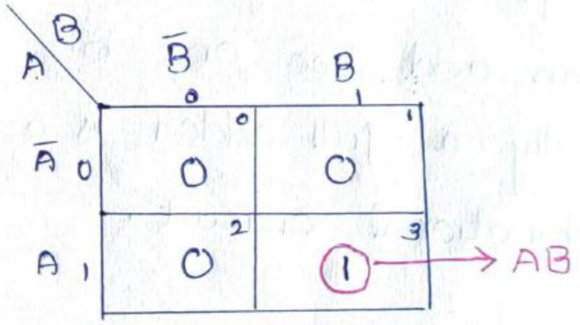
fig:- Block diagram of Half Adder

ii, Truth Table

S.NO	Inputs		Outputs	
	A	B	Sum	Carry
1	0	0	0	0
2	0	1	1	0
3	1	0	1	0
4	1	1	0	1

iii, K-maps for Carry and Sum outputs





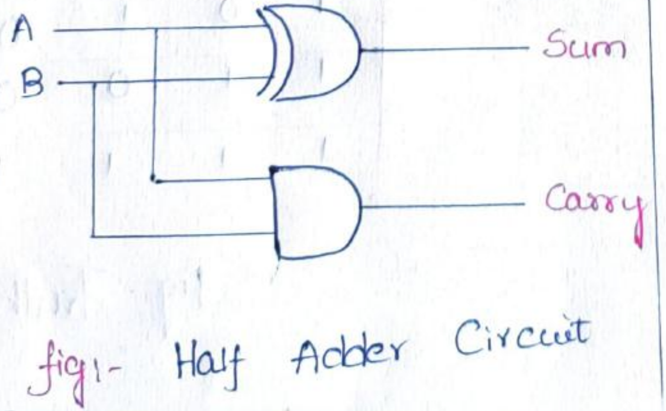
Carry = AB

$\text{Sum} = \bar{A}B + A\bar{B} = A \oplus B$

fig:- K-map for Sum output

fig:- K-map for Carry output

iv, Implementation of logic diagram



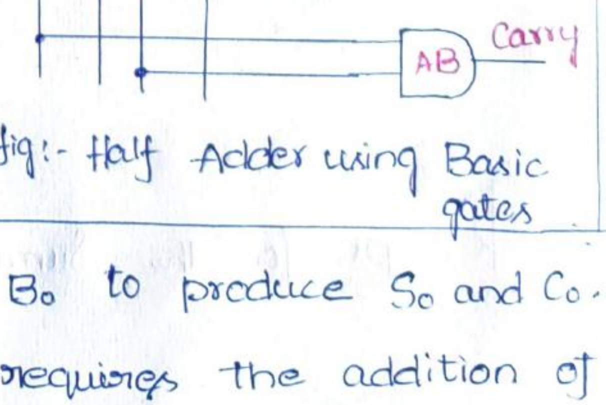


fig:- Half Adder Circuit

fig:- Half Adder using Basic gates

Drawback:-

A half adder can add A_0 and B_0 to produce S_0 and C_0. However, the addition of next bits requires the addition of

A_1, B_1 and C_0. The addition of three bits is not possible to perform by using an half adder circuit. Therefore, practically we cannot use a half adder.

$$\begin{array}{l} A = A_1 A_0 \\ B = B_1 B_0 \\ \underline{+ \; C_0} \xrightarrow{\quad} \text{Carry generated from the addition } (A_0 + B_0) \\ \underline{C_1 \; S_1 \; S_0} \end{array}$$

Full Adder:

To overcome the drawback of an Half Adder circuit, we develop a 3 single bit adder circuit called Full Adder. It can add two one-bit numbers A and B, and carry C_{in}. Basically, a full adder is a three input and two output combinational circuit.

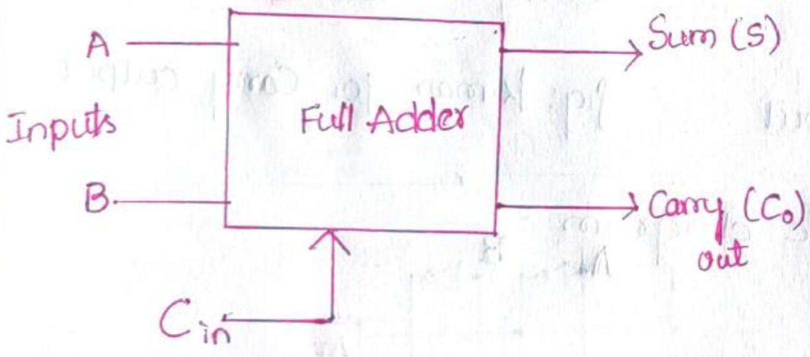


fig1- Block diagram

Inputs			Outputs	
A	B	C_{in}	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

fig:- Truth Table

K-maps for the Sum(S) and Carry out (C_0) outputs.

K-map for Sum

$A \backslash BC_{in}$	$\bar{B}\bar{C}_{in}$	$\bar{B}C_{in}$	BC_{in}	$B\bar{C}_{in}$
$\bar{A}$	0 ^{0}	1 ^{1}	0 ^{3}	1 ^{2}
A	1 ^{4}	0 ^{5}	1 ^{7}	0 ^{6}

$\bar{A}\bar{B}C_{in}$ $\quad$ $\bar{A}B\bar{C}_{in}$ $\quad$ $A\bar{B}\bar{C}_{in}$ $\quad$ ABC_{in}

K-map for Carry out

$A \backslash BC_{in}$	$\bar{B}\bar{C}_{in}$	$\bar{B}C_{in}$	BC_{in}	$B\bar{C}_{in}$
$\bar{A}$	0 ^{0}	0 ^{1}	1 ^{3}	0 ^{2}
A	0 ^{4}	1 ^{5}	1 ^{7}	1 ^{6}

BC_{in} $\quad$ AC_{in} $\quad$ AB

Carry out $= BC_{in} + AC_{in} + AB$

$$Sum = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$$

$$= C_{in}(\bar{A}\bar{B} + AB) + \bar{C}_{in}(\bar{A}B + A\bar{B})$$

$$= C_{in}(\overline{A \oplus B}) + \bar{C}_{in}(A \oplus B)$$

Let $x = A \oplus B$

$$Sum = C_{in}(\bar{x}) + \bar{C}_{in}(x)$$

$$= x \oplus C_{in}$$

$\therefore Sum = A \oplus B \oplus C_{in}$

Logic Diagram for full Adder

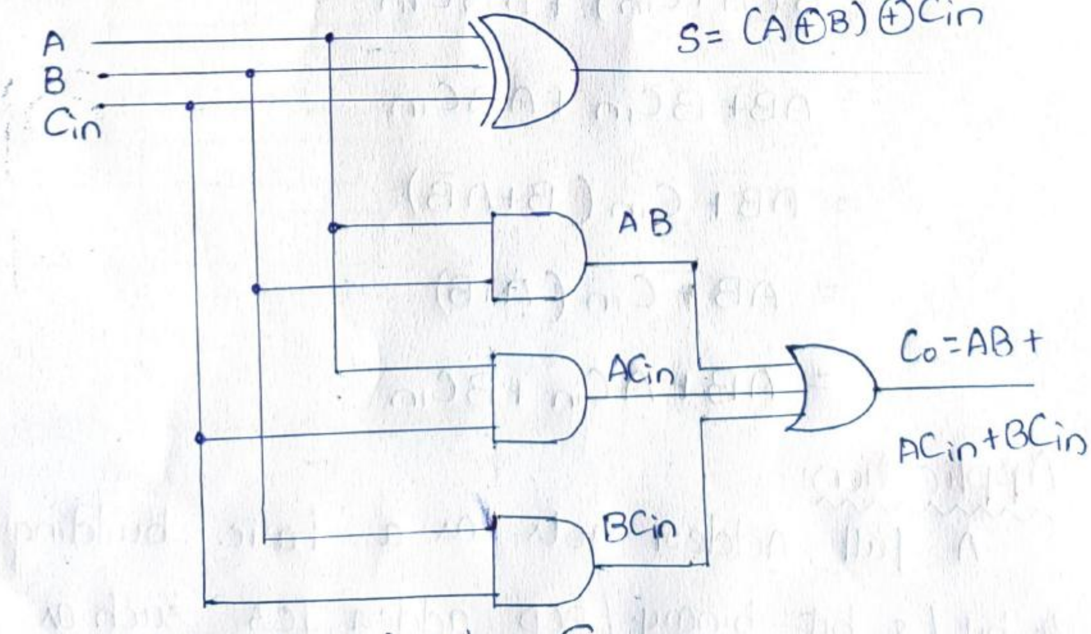


fig:- Full Adder Circuit

Full Adder Using Half Adders:-

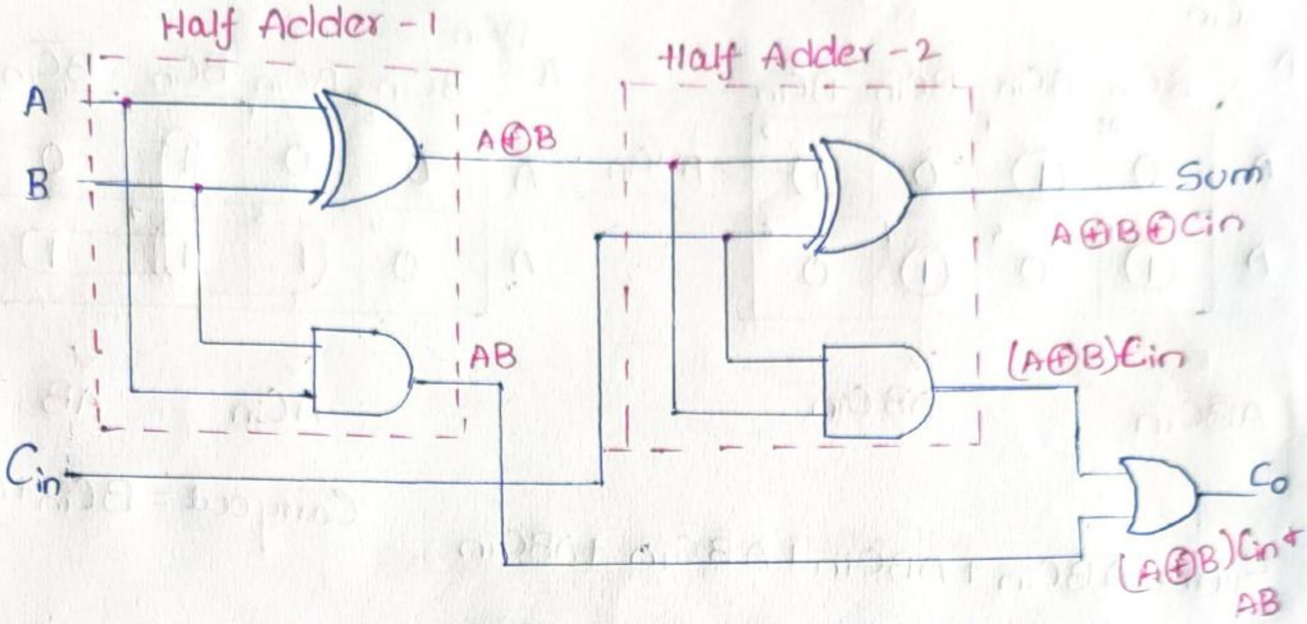


fig:- Full Adder using two Half Adders

$Sum = A \oplus B \oplus C_{in}$

$C_o = (A \oplus B)C_{in} + AB$

$= (\bar{A}B + A\bar{B})\,C_{in} + AB$

$= \bar{A}BC_{in} + A\bar{B}C_{in} + AB$

$= B(\bar{A}C_{in} + A) + A\bar{B}C_{in}$ $\quad [\because A + \bar{A}B = A + B]$

$= B(A + C_{in}) + A\bar{B}C_{in}$

$= AB + BC_{in} + A\bar{B}C_{in}$

$= AB + C_{in}(B + A\bar{B})$

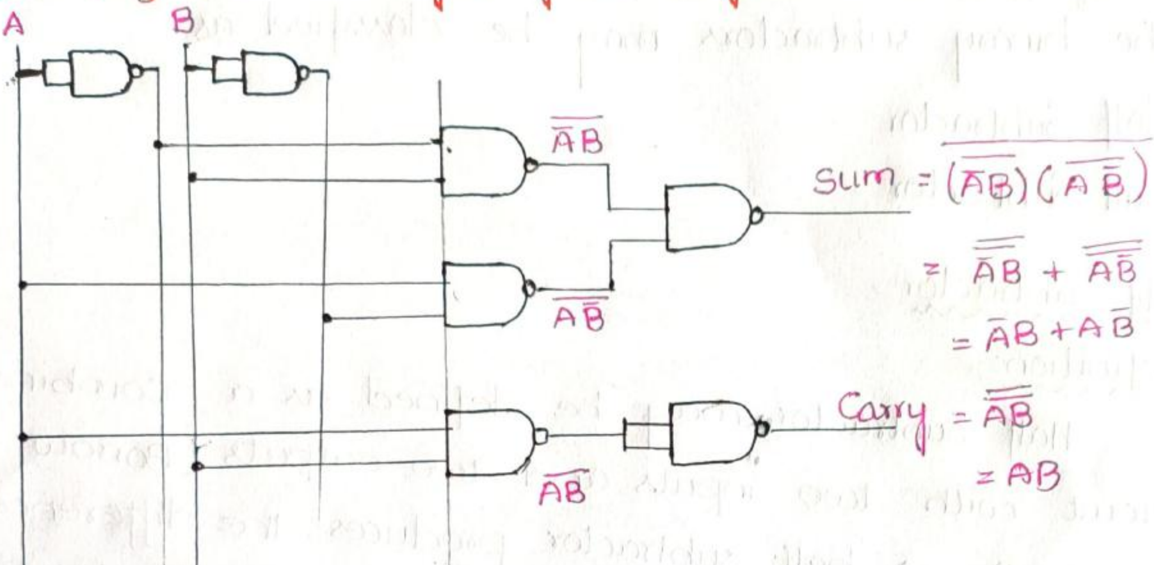
$= AB + C_{in}(A+B)$

$= AB + AC_{in} + BC_{in}$

Application:-

A full Adder acts as a basic building block of the 4 bit / 8 bit binary / BCD adder ICs such as 7483.

1. Implement Half Adder using only NAND gates.



$$\text{Sum} = \overline{(\overline{\bar{A}B})(\overline{A\bar{B}})}$$
$$= \overline{\overline{\bar{A}B}} + \overline{\overline{A\bar{B}}}$$
$$= \bar{A}B + A\bar{B}$$

$$\text{Carry} = \overline{\overline{AB}}$$
$$= AB$$

fig:- Half Adder Using only NAND gates.

2. Implement Full Adder using only NAND gates

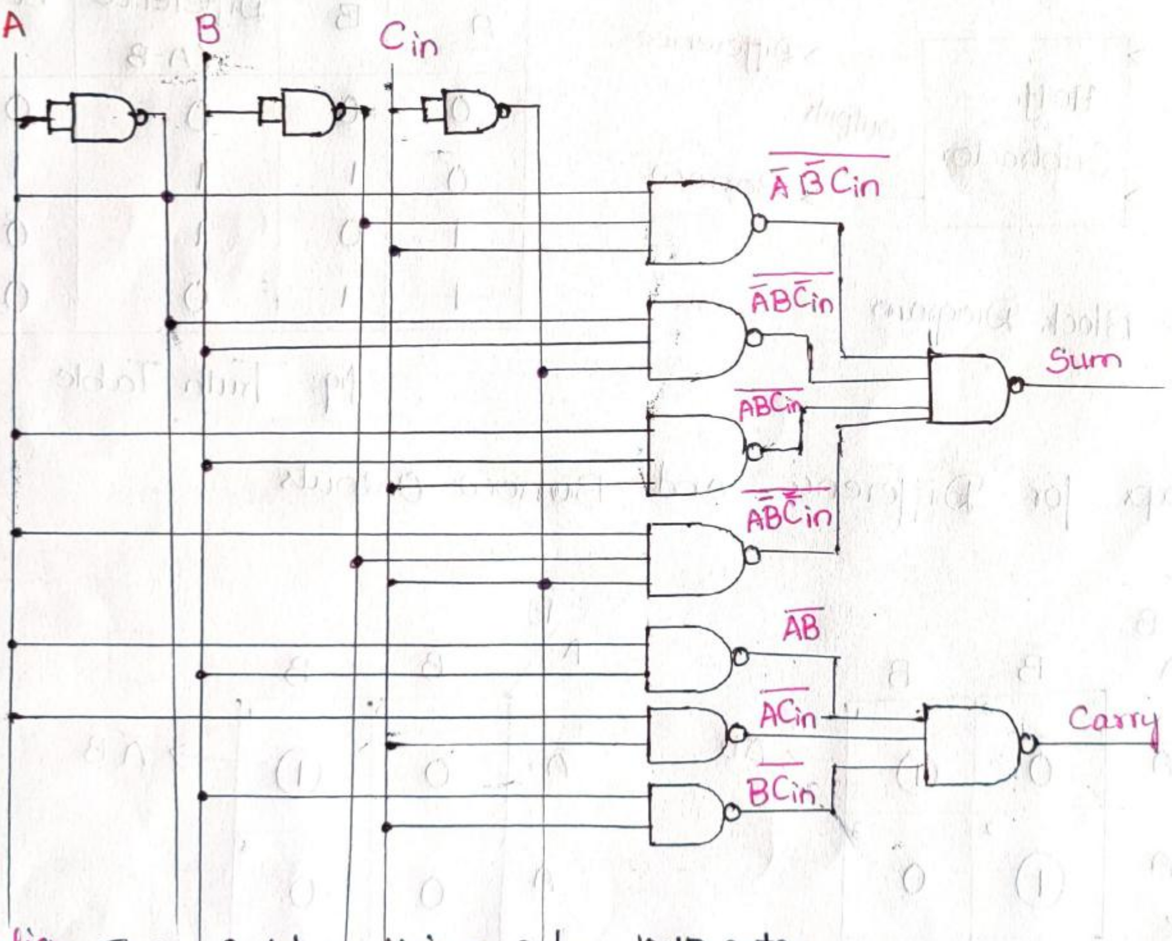

A
B
C_{in}
$\overline{\bar{A}\bar{B}C_{in}}$
$\overline{\bar{A}B\bar{C}_{in}}$
$\overline{ABC_{in}}$
$\overline{A\bar{B}\bar{C}_{in}}$
Sum
$\overline{AB}$
$\overline{AC_{in}}$
$\overline{BC_{in}}$
Carry

fig:- Full Adder Using only NAND gates

Binary Subtractors

The binary subtractors may be classified as:

- Half Subtractor
- Full Subtractor

Half Subtractor:-

Definition:-

Half subtractor may be defined as a combinational circuit with two inputs and two outputs (Borrow and difference). A half subtractor produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed.

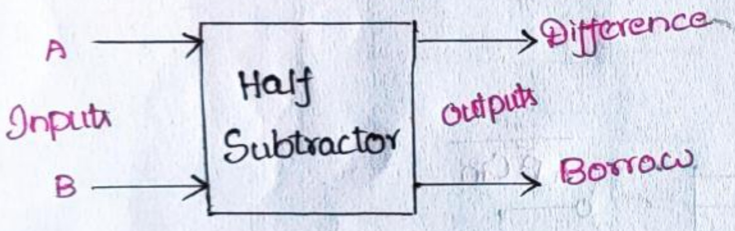
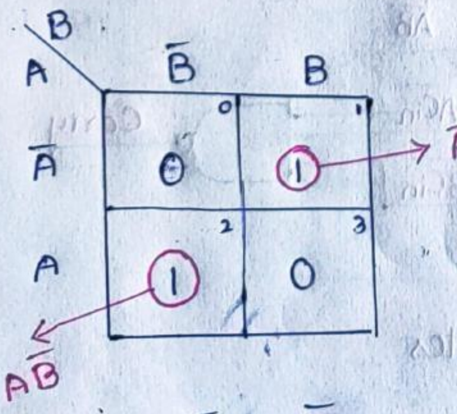


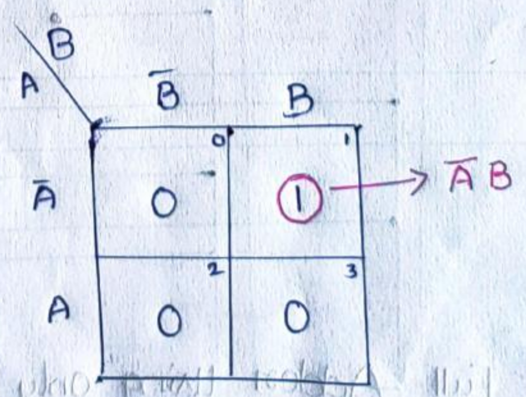
fig:- Block Diagram

Inputs		Outputs	
A	B	Difference (A-B	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

fig:- Truth Table

K-maps for Difference and Borrow Outputs





Difference $= \bar{A}B + A\bar{B}$

$= A \oplus B$

Borrow $= \bar{A}B$

fig:- k-map for Difference

fig:- k-map for Borrow

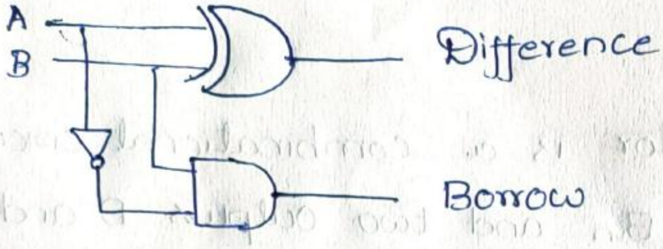


fig :- Half subtractor circuit

Drawback

An half subtractor can only perform the subtraction of two binary bits. However, while performing the subtraction, it does not take into account the borrow of the lower significant stage.

Implementation of Half Subtractor using Basic Gates.

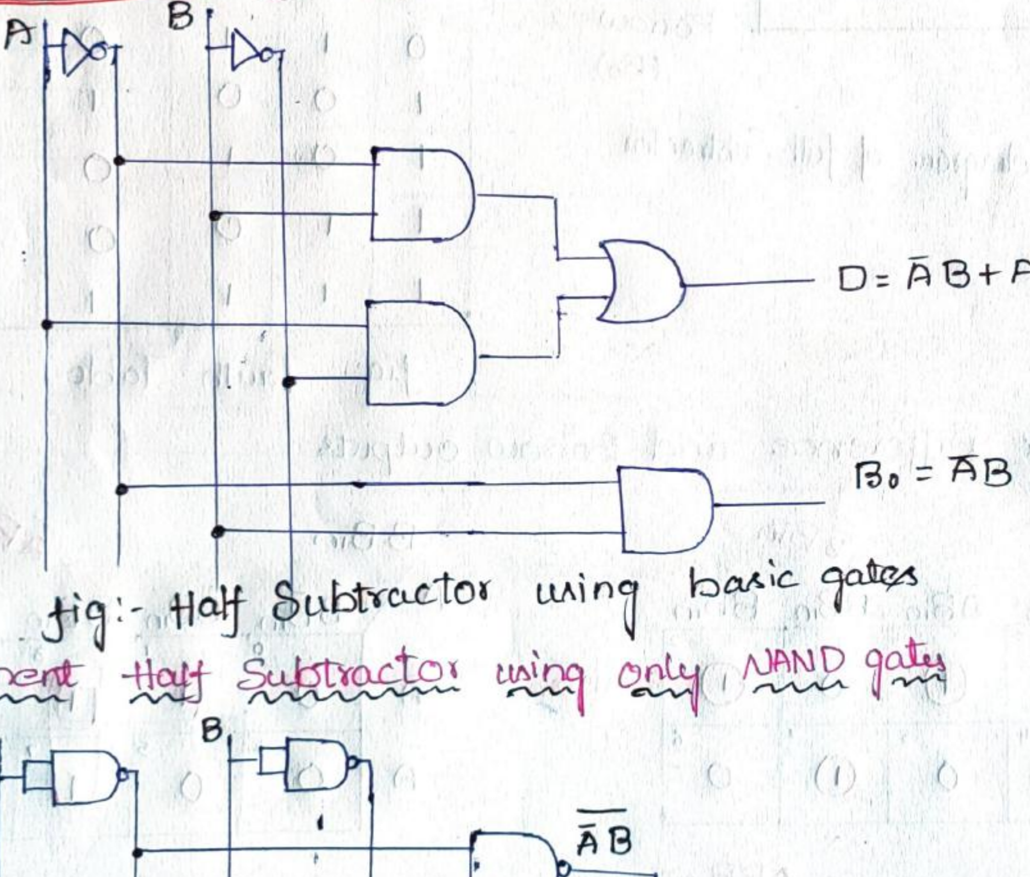


fig :- Half Subtractor using basic gates

Implement Half Subtractor using only NAND gates

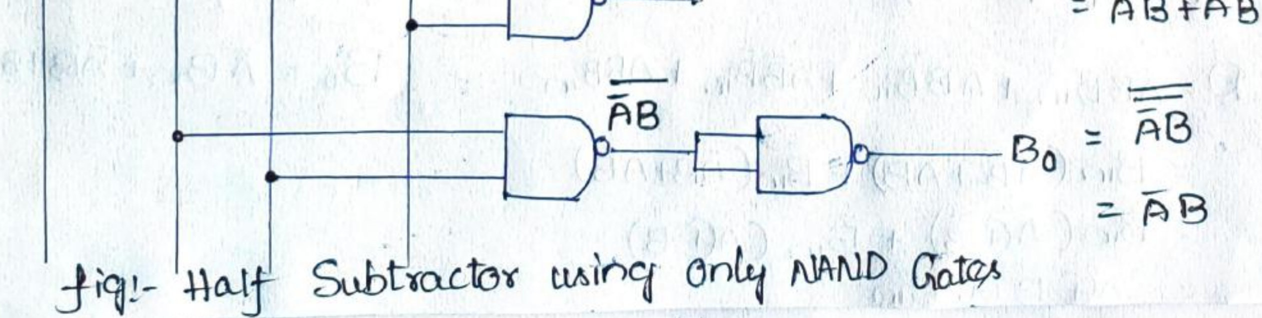


fig :- Half Subtractor using Only NAND Gates

Full Subtractor

Definition:-

A full subtractor is a combinational circuit with three inputs A, B and Bin and two outputs D and B_o. Here, A is the minuend, B is subtrahend, B_{in} is the borrow produced by the previous stage, D is the difference output and B_o is the Borrow output.

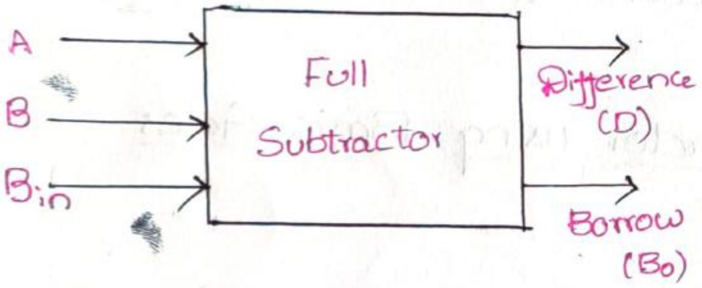


fig:- Block diagram of full Subtractor

Inputs			Outputs	
A	B	B_{in}	D	B_o
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

fig:- Truth Table

K-maps for Difference and Borrow outputs

A \ BB_{in}	$\bar{B}\bar{B}_{in}$	$\bar{A}B_{in}$	BB_{in}	$B\bar{B}_{in}$
$\bar{A}$	0 0	1 1	3 0	2 1
A	4 1	5 0	7 1	6 0

$\bar{A}\bar{B}B_{in}$

$\bar{A}B\bar{B}_{in}$

$A\bar{B}\bar{B}_{in}$

ABB_{in}

~~ff~~ map for Difference

A \ BB_{in}	$\bar{B}\bar{B}_{in}$	$\bar{B}B_{in}$	BB_{in}	$B\bar{B}_{in}$
$\bar{A}$	0 0	1 1	3 1	2 1
A	4 0	5 0	7 1	6 0

$\bar{A}B_{in}$

$\bar{A}B$

BB_{in}

k.map for Borrow

$B_o = \bar{A}B_{in} + \bar{A}B + BB_{in}$

$D = \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in} + ABB_{in}$

$= B_{in}(\bar{A}\bar{B} + AB) + \bar{B}_{in}(\bar{A}B + A\bar{B})$

$= B_{in}(\overline{A \oplus B}) + \bar{B}_{in}(A \oplus B)$

$= A \oplus B \oplus B_{in}$

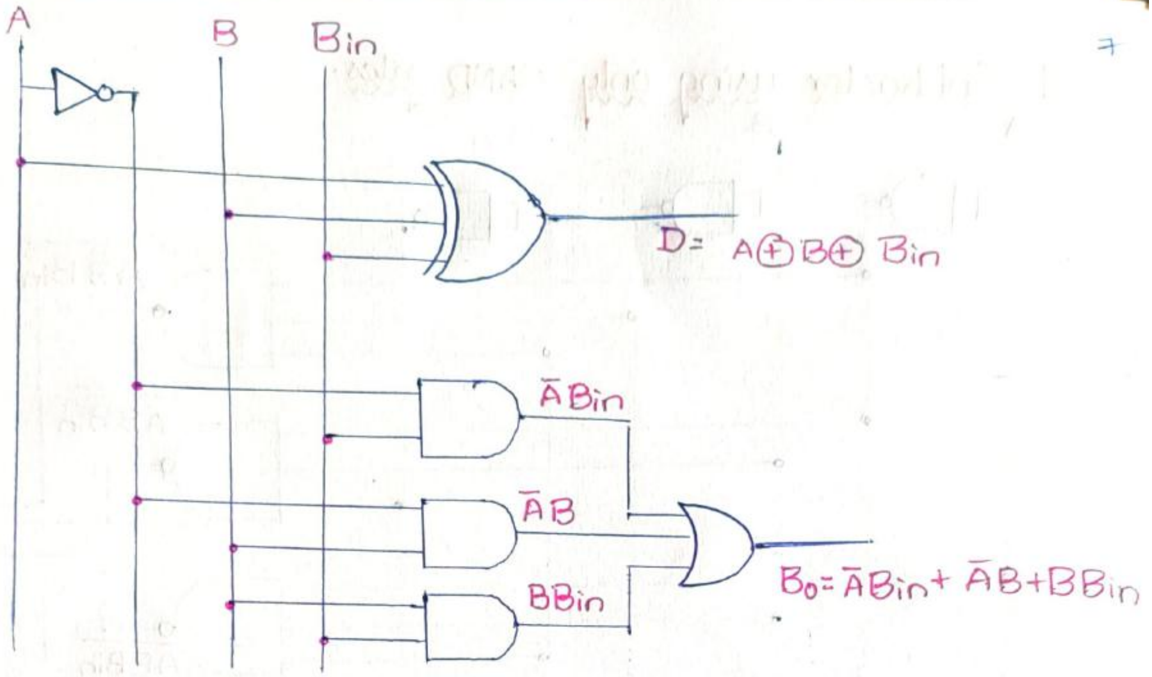

A
B
Bin
$D = A \oplus B \oplus Bin$
$\bar{A}Bin$
$\bar{A}B$
$BBin$
$B_0 = \bar{A}Bin + \bar{A}B + BBin$

fig:- Logic Diagram for a full Subtractor

Full Subtractor using Half Subtractors.

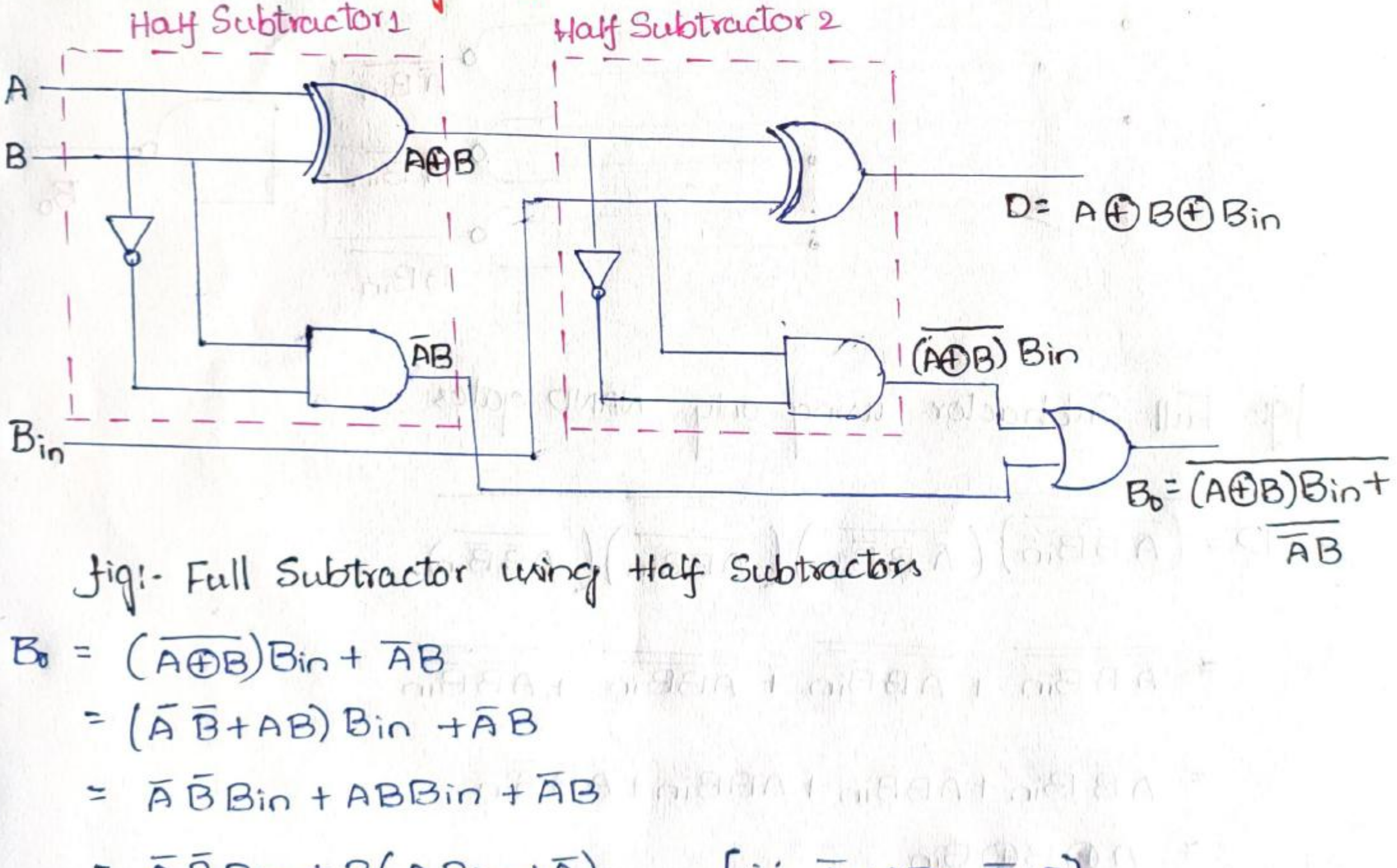

Half Subtractor1
Half Subtractor 2
A
B
$A \oplus B$
$D = A \oplus B \oplus Bin$
$\bar{A}B$
$\overline{(A \oplus B)}\, Bin$
B_{in}
$B_0 = \overline{(A \oplus B)}Bin + \bar{A}B$

fig:- Full Subtractor using Half Subtractors

$$
\begin{aligned}
B_0 &= \overline{(A \oplus B)}Bin + \bar{A}B \\
&= (\bar{A}\,\bar{B} + AB)\,Bin + \bar{A}B \\
&= \bar{A}\,\bar{B}\,Bin + ABBin + \bar{A}B \\
&= \bar{A}\,\bar{B}\,Bin + B(A\,Bin + \bar{A}) \qquad [\because \bar{A} + AB = \bar{A} + B] \\
&= \bar{A}\,\bar{B}\,Bin + B(\bar{A} + Bin) \\
&= \bar{A}\bar{B}\,Bin + \bar{A}B + BBin \\
&= \bar{A}\,(\bar{B}Bin + B) + BBin \\
&= \bar{A}\,(B + Bin) + BBin \\
&= \bar{A}B + \bar{A}\,Bin + BBin
\end{aligned}
$$

Full Subtractor using only NAND gates.

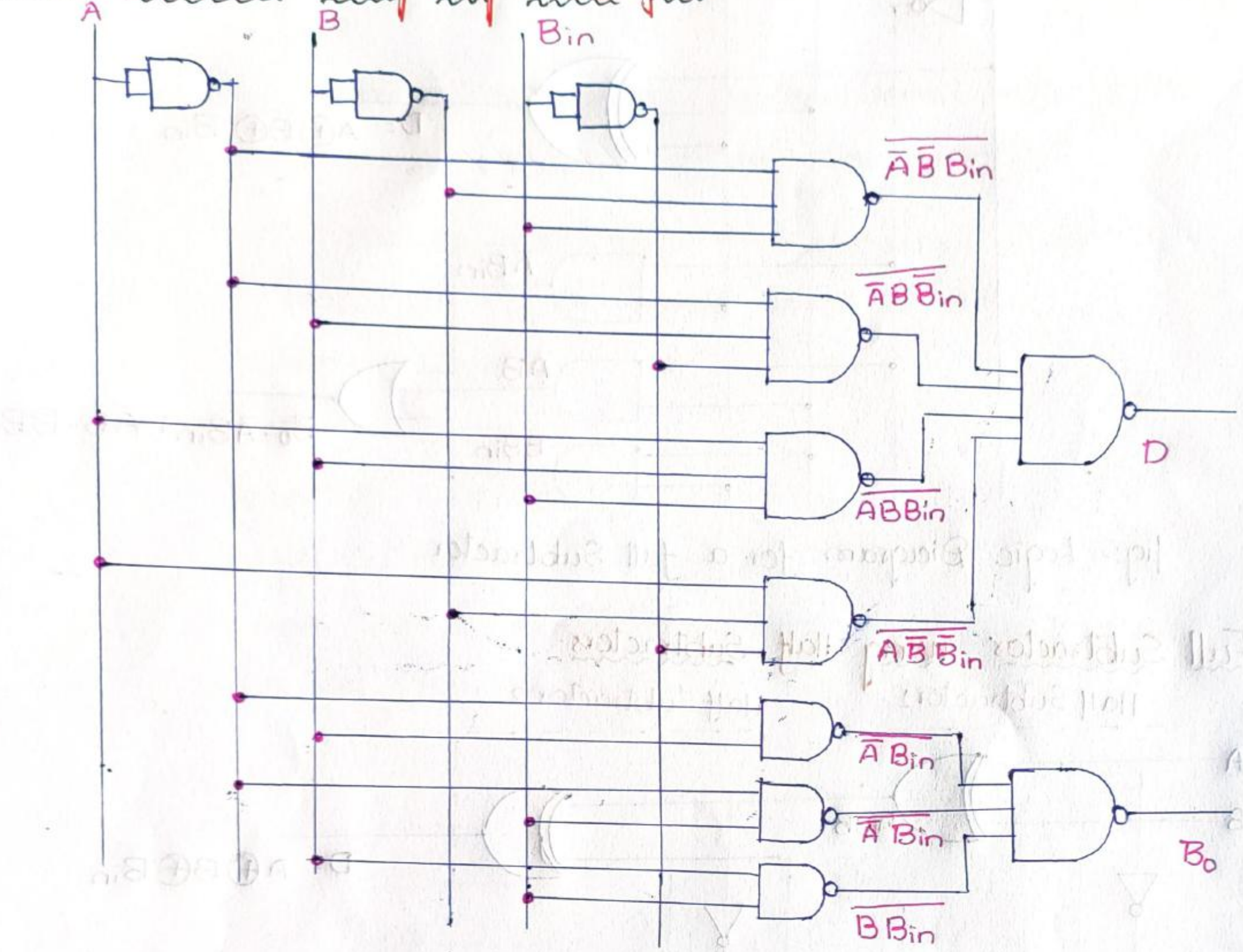

A
B
Bin
$\overline{\bar{A}\bar{B}B_{in}}$
$\overline{\bar{A}B\bar{B}_{in}}$
$\overline{ABB_{in}}$
$\overline{A\bar{B}\bar{B}_{in}}$
D
$\overline{\bar{A}B_{in}}$
$\overline{\bar{A}B_{in}}$
$\overline{BB_{in}}$
B_o

fig:- Full Subtractor using only NAND gates

$$D = \overline{\left(\overline{\bar{A}\bar{B}B_{in}}\right)\left(\overline{\bar{A}B\bar{B}_{in}}\right)\left(\overline{ABB_{in}}\right)\left(\overline{A\bar{B}\bar{B}_{in}}\right)}$$

$$= \overline{\overline{\bar{A}\bar{B}B_{in}}} + \overline{\overline{\bar{A}B\bar{B}_{in}}} + \overline{\overline{ABB_{in}}} + \overline{\overline{A\bar{B}\bar{B}_{in}}}$$

$$= \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + ABB_{in} + A\bar{B}\bar{B}_{in}$$

$$= A \oplus B \oplus B_{in}$$

$$B_o = \overline{\left(\overline{\bar{A}B}\right)\left(\overline{\bar{A}B_{in}}\right)\left(\overline{BB_{in}}\right)}$$

$$= \overline{\overline{\bar{A}B}} + \overline{\overline{\bar{A}B_{in}}} + \overline{\overline{BB_{in}}}$$

$$= \bar{A}B + \bar{A}B_{in} + BB_{in}$$

MULTIPLEXERS:

A Multiplexers (MUX) is a combinational logic component that has several inputs and only one output.

- Mux directs one of the inputs to its output line by using a control bit word to its select lines.
- MUX contains
  - * 2^n inputs
  - * n selection inputs
  - * a single output
  - * Selection input determines the input that should be connected to the output.

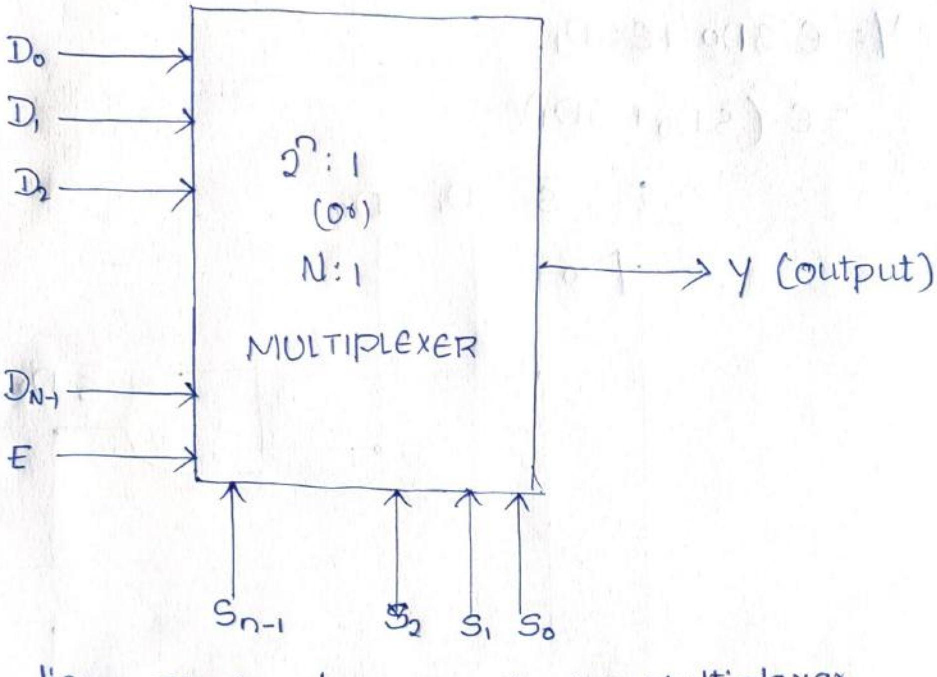


fig:- Block diagram of N:1 Multiplexer

- The Multiplexer is also called as data selector.
- The Multiplexer acts like an electronic switch that selects one from different.
- A multiplexer may have an earble input to control the operation of the Unit.

2:1 MULTIPLEXER

2:1 Multiplexer has two data inputs D_0 and D_1, one Select input S, an enable input and one output. The block diagram of 2:1 multiplexer is shown in figure.

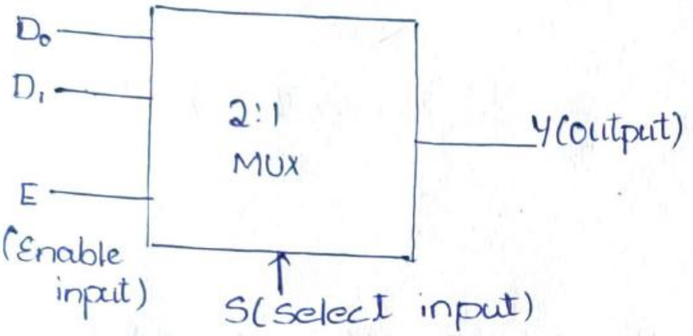


fig:- Block Diagram

Enable	Select i/p S	Output Y
0	x	0
1	0	D_0
1	1	D_1

Truth Table

$$Y = E\bar{S}D_0 + ESD_1$$
$$= E(\bar{S}D_0 + SD_1)$$

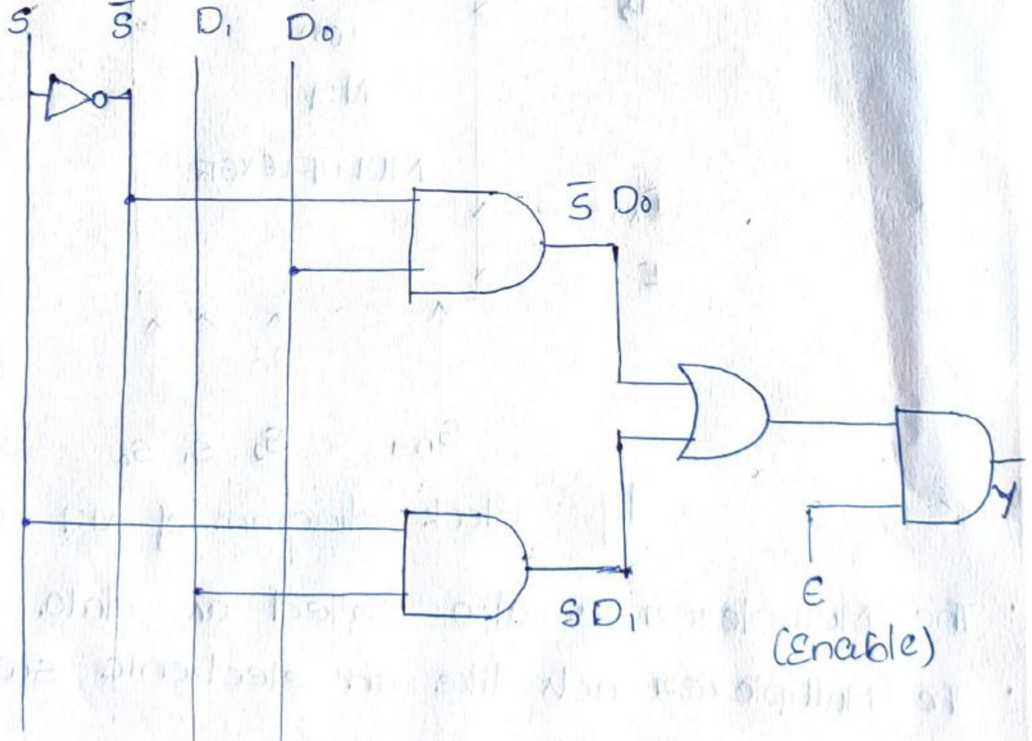


fig:- Realization of 2:1 MUX using gates

4:1 MULTIPLEXER:

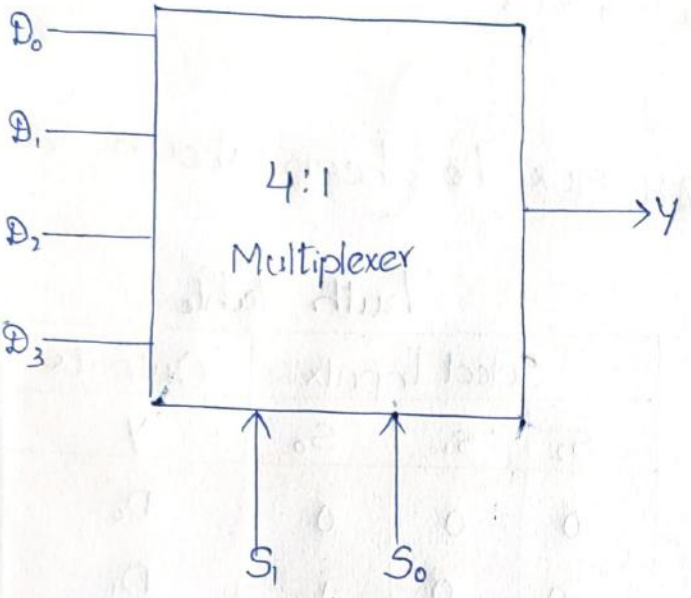


fig:- Block diagram

Select inputs		Output
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Truth table

It has four data inputs, two select lines and one output.

Truth table tells us

$Y = D_0$ when $S_1 S_0 = 00$

$Y = D_1$ when $S_1 S_0 = 01$

$Y = D_2$ when $S_1 S_0 = 10$

$Y = D_3$ when $S_1 S_0 = 11$

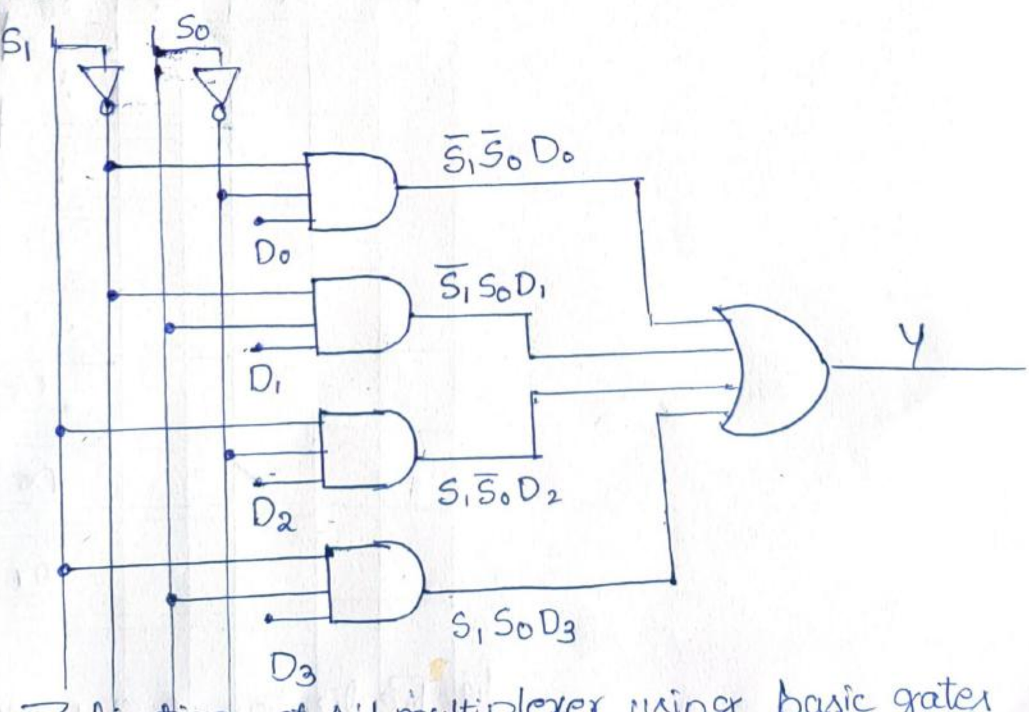


fig:- Realization of 4:1 multiplexer using basic gates

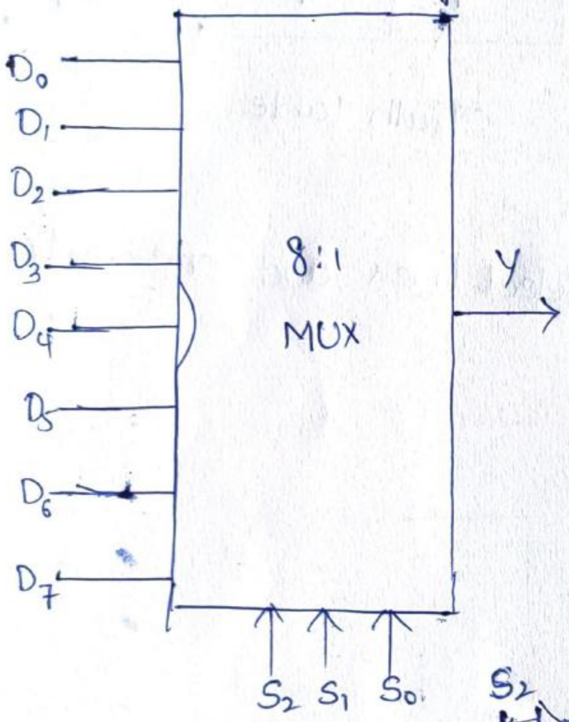
The output will be high when the selected input is 1. Hence, the logical expression for output in the SOP form will be

$$Y = \overline{S_1}\,\overline{S_0}\,D_0 + \overline{S_1}\,S_0\,D_1 + S_1\,\overline{S_0}\,D_2 + S_1 S_0 D_3$$

8:1 MULTIPLEXER:-

The block diagram of an 8:1 MUX has been shown in below figure and its Truth table.

Block Diagram



Truth Table

Select Inputs			Outputs
S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

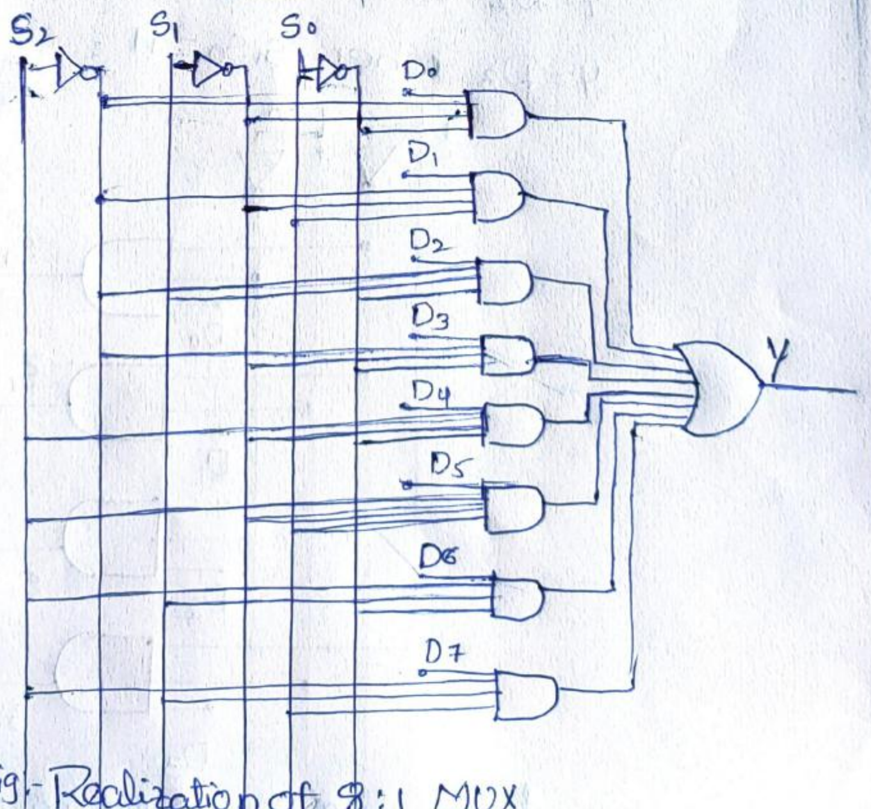


fig:- Realization of 8:1 MUX

Applications of a Multiplexer:-

- It is used as a data selector to select one out of many data inputs.
- It is used for simplification of logic design.
- In the data acquisition system.
- In designing the combinational circuits.
- In the D/A converters
- To minimize the number of connections.

Multiplexer Tree (Expanding Multiplexers)

The multiplexers having more number of inputs can be obtained by cascading two or more multiplexers with less number of inputs. This is known as a multiplexer tree.

1. Implement an 8:1 multiplexer using two 4:1 multiplexers.

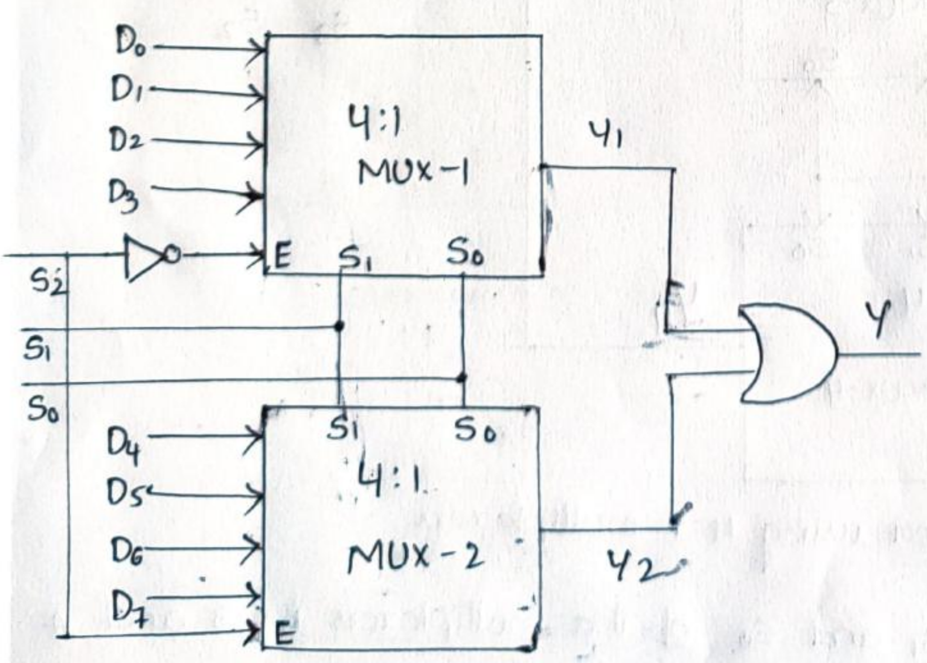


fig1- Multiplexer by cascading two 4:1 Multiplexers.

Select inputs			output	
S_2	S_1	S_0	Y	
0	0	0	D_0	MUX-1 is Enable
0	0	1	D_1	
0	1	0	D_2	
0	1	1	D_3	
1	0	0	D_4	MUX-2 is Enable
1	0	1	D_5	
1	1	0	D_6	
1	1	1	D_7	

Truth Table

2. Implement a 16:1 multiplexer using 4:1 multiplexers.

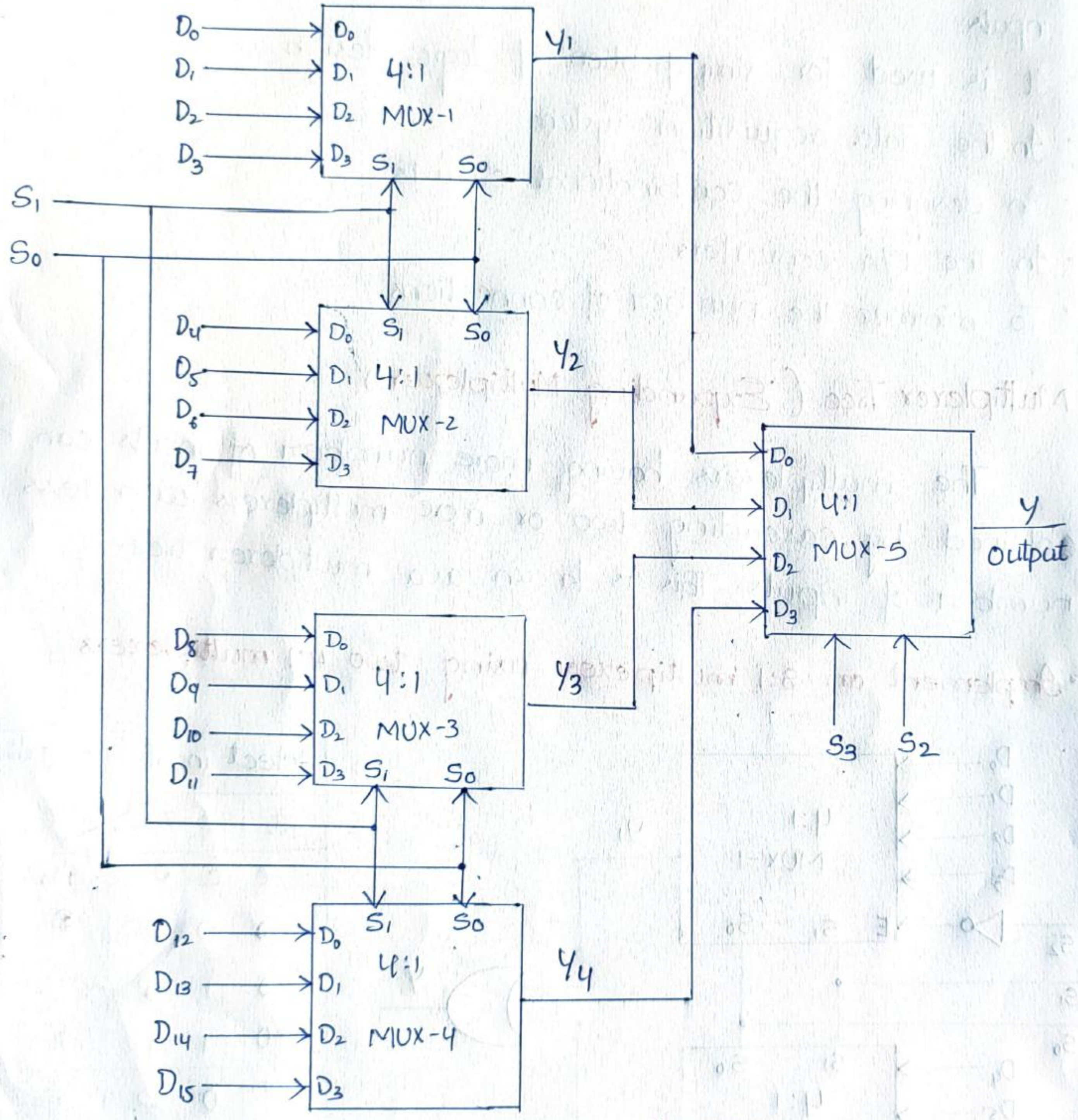


fig:- 16:1 Multiplexer using 4:1 multiplexers

The Select input S_1 and S_0 of the multiplexers 1, 2, 3 and 4 are connected together. The select inputs S_3 and S_2 are applied to the data inputs D_0, D_1, D_2 and D_3 of MUX-5 as shown in figure.

Select inputs				MUX · Outputs				Final Output	
S_3	S_2	S_1	S_0	Y_1	Y_2	Y_3	Y_4	Y	
0	0	0	0	D_0	D_4	D_8	D_{12}	D_0	$S_3 S_2 = 00$ MUX-5 Selects Y_1
0	0	0	1	D_1	D_5	D_9	D_{13}	D_1	
0	0	1	0	D_2	D_6	D_{10}	D_{14}	D_2	
0	0	1	1	D_3	D_7	D_{11}	D_{15}	D_3	
0	1	0	0	D_0	D_4	D_8	D_{12}	D_4	$S_3 S_2 = 01$ MUX-5 Selects Y_2
0	1	0	1	D_1	D_5	D_9	D_{13}	D_5	
0	1	1	0	D_2	D_6	D_{10}	D_{14}	D_6	
0	1	1	1	D_3	D_7	D_{11}	D_{15}	D_7	
1	0	0	0	D_0	D_4	D_8	D_{12}	D_8	$S_3 S_2 = 10$ MUX-5 Selects Y_3
1	0	0	1	D_1	D_5	D_9	D_{13}	D_9	
1	0	1	0	D_2	D_6	D_{10}	D_{14}	D_{10}	
1	0	1	1	D_3	D_7	D_{11}	D_{15}	D_{11}	
1	1	0	0	D_0	D_4	D_8	D_{12}	D_{12}	$S_3 S_2 = 11$ MUX-5 Selects Y_4
1	1	0	1	D_1	D_5	D_9	D_{13}	D_{13}	
1	1	1	0	D_2	D_6	D_{10}	D_{14}	D_{14}	
1	1	1	1	D_3	D_7	D_{11}	D_{15}	D_{15}	

Table:- Summary of Operation.

Select inputs				MUX·Outputs				Final output Y	
S_3	S_2	S_1	S_0	Y_1	Y_2	Y_3	Y_4		
0	0	0	0	D_0	D_4	D_8	D_{12}	D_0	$S_3 S_2 = 00$ MUX-5 Selects Y_1
0	0	0	1	D_1	D_5	D_9	D_{13}	D_1	
0	0	1	0	D_2	D_6	D_{10}	D_{14}	D_2	
0	0	1	1	D_3	D_7	D_{11}	D_{15}	D_3	
0	1	0	0	D_0	D_4	D_8	D_{12}	D_4	$S_3 S_2 = 01$ MUX-5 Selects Y_2
0	1	0	1	D_1	D_5	D_9	D_{13}	D_5	
0	1	1	0	D_2	D_6	D_{10}	D_{14}	D_6	
0	1	1	1	D_3	D_7	D_{11}	D_{15}	D_7	
1	0	0	0	D_0	D_4	D_8	D_{12}	D_8	$S_3 S_2 = 10$ MUX-5 Selects Y_3
1	0	0	1	D_1	D_5	D_9	D_{13}	D_9	
1	0	1	0	D_2	D_6	D_{10}	D_{14}	D_{10}	
1	0	1	1	D_3	D_7	D_{11}	D_{15}	D_{11}	
1	1	0	0	D_0	D_4	D_8	D_{12}	D_{12}	$S_3 S_2 = 11$ MUX-5 Selects Y_4
1	1	0	1	D_1	D_5	D_9	D_{13}	D_{13}	
1	1	1	0	D_2	D_6	D_{10}	D_{14}	D_{14}	
1	1	1	1	D_3	D_7	D_{11}	D_{15}	D_{15}	

Table:- Summary of Operation.

4. Implement a full adder using 8:1 multiplexer

The truth table of a full adder is

Inputs			Outputs	
A	B	Cin	Sum (S)	Carry (C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The Sum and Carry outputs can be expressed in the standard SOP form is

$S = \Sigma m(1,2,4,7)$ and $C = \Sigma m(3,5,6,7)$

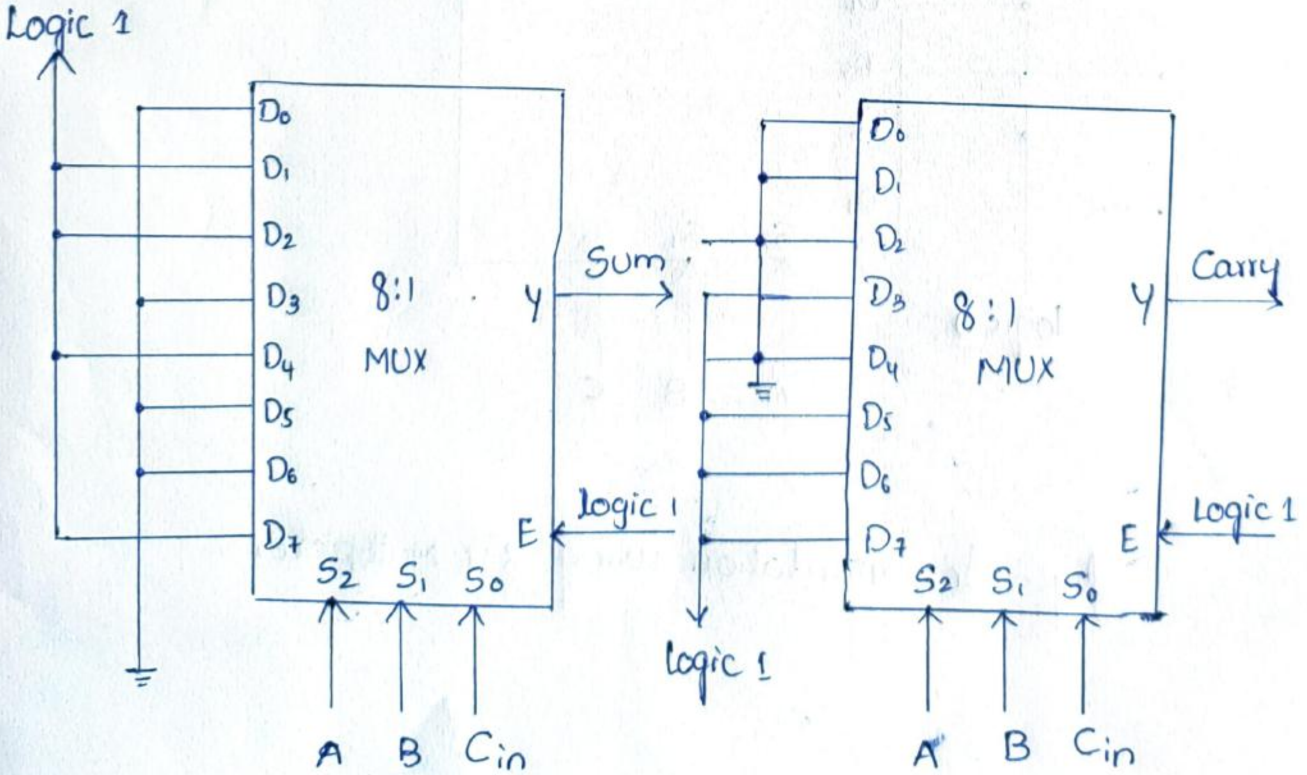


fig:- Full Adder using 8:1 MUX

Multiplex

5. Implement the following function using an 8:1 multiplexer.

$$f(A,B,C,D) = \pi M(0,2,4,6,8,10,12,14)$$

Inputs	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	0	①	2	③	4	⑤	6	⑦
A	8	⑨	10	⑪	12	⑬	14	⑮
Input to MUX	0	1	0	1	0	1	0	1

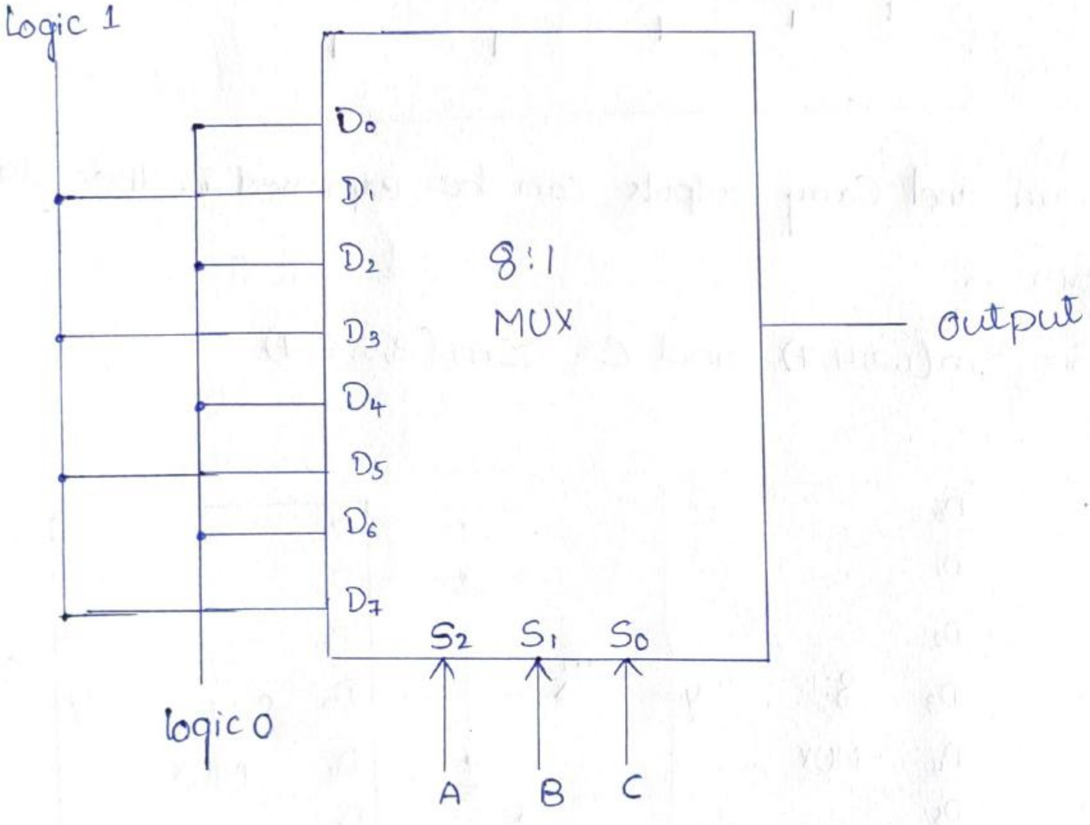


fig:- Implementation using 8:1 Multiplexer

DEMULTIPLEXER

→ It is also called a "data distributor".

→ A demultiplexer is a device that takes a single input line and routes it one of several digital output lines.

→ A demultiplexer has 2^n outputs and n select lines which are used to select which input line to send to the output.

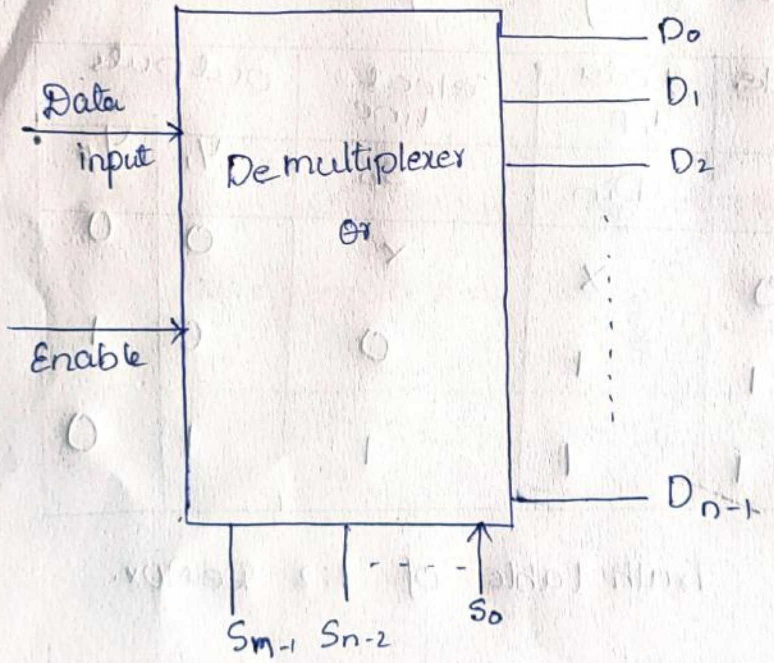


fig:- 1:n demultiplexer

1:2 Demux

A 1:2 demultiplexer has one data input D_{in}, One Select input S_0, One Enable (E) input and two outputs Y_0 and Y_1.

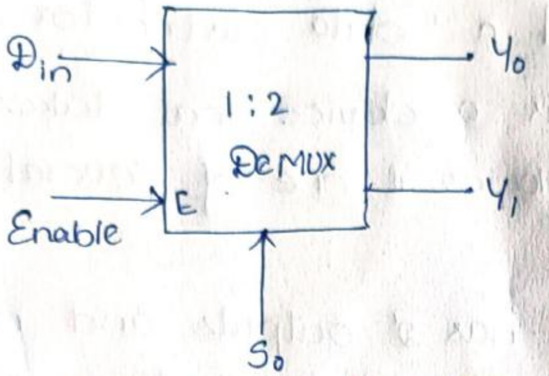


fig:- Block diagram

- If Enable =0, then all the outputs are 0.
- If Enable input =1, one of the outputs Y_0 (or) Y_1 is active for a given input.

Enable	Data i/p	Select line	outputs	
			Y_1	Y_0
E	Din	S_0		
0	X	X	0	0
1	1	0	0	1
1	1	1	1	0

Truth table of 1:2 DeMUX.

$Y_0 = E D_{in} \overline{S_0}$

$Y_1 = E D_{in} S_0$

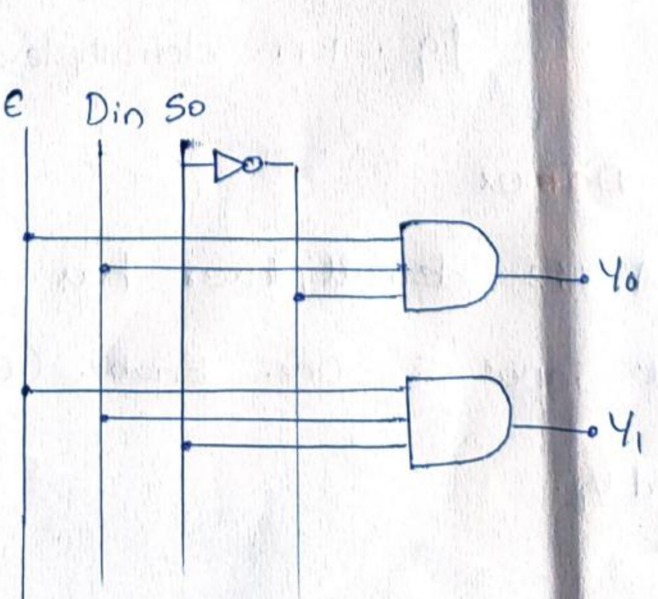


fig:- 1:2 DeMux using gates

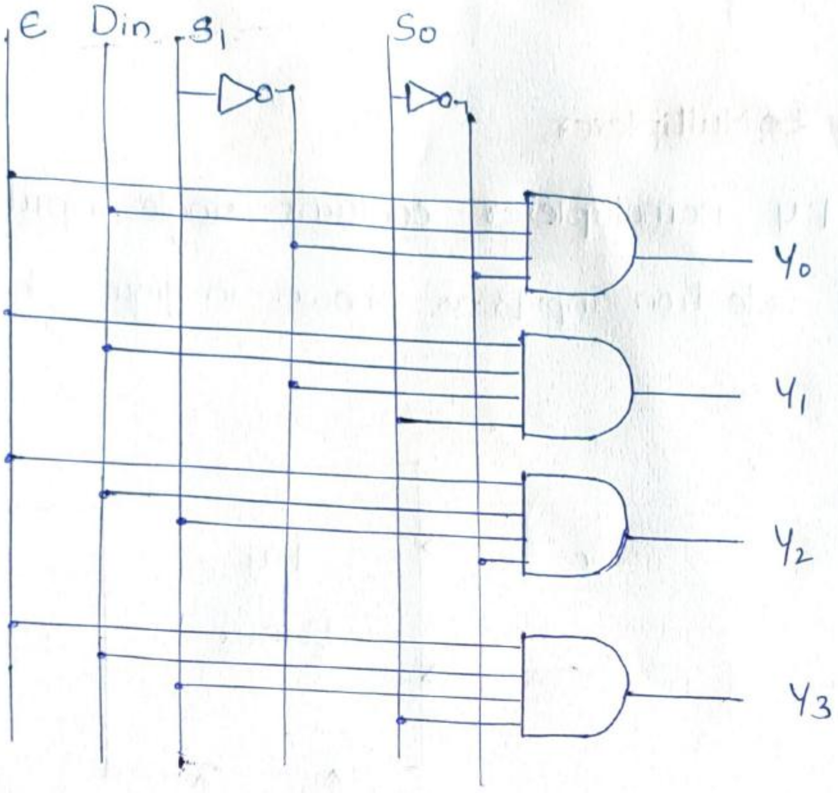


fig:- 1:4 Demultiplexer.

1:8 Demultiplexer

1:8 Demultiplexer has one data input, eight outputs, three Select inputs and an enable input E As shown in block diagram.

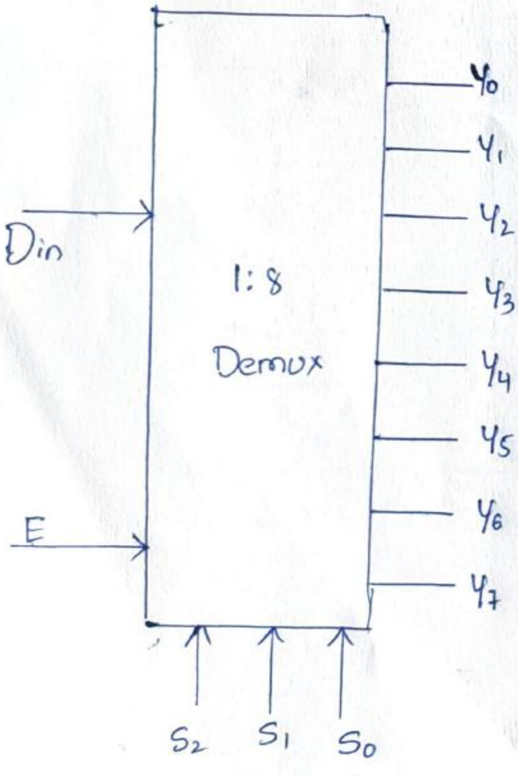


fig:- Block diagram of 1:8 Demux

Enable	Select			Outputs							
E	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	Din
1	0	0	1	0	0	0	0	0	0	Din	0
1	0	1	0	0	0	0	0	0	Din	0	0
1	0	1	1	0	0	0	0	Din	0	0	0
1	1	0	0	0	0	0	Din	0	0	0	0
1	1	0	1	0	0	Din	0	0	0	0	0
1	1	1	0	0	Din	0	0	0	0	0	0
1	1	1	1	Din	0	0	0	0	0	0	0

Truth table of 1:8 Demux

1:4 DeMultiplexer

1:4 Demultiplexer contains single input, four outputs and two Selection inputs as shown in figure below.

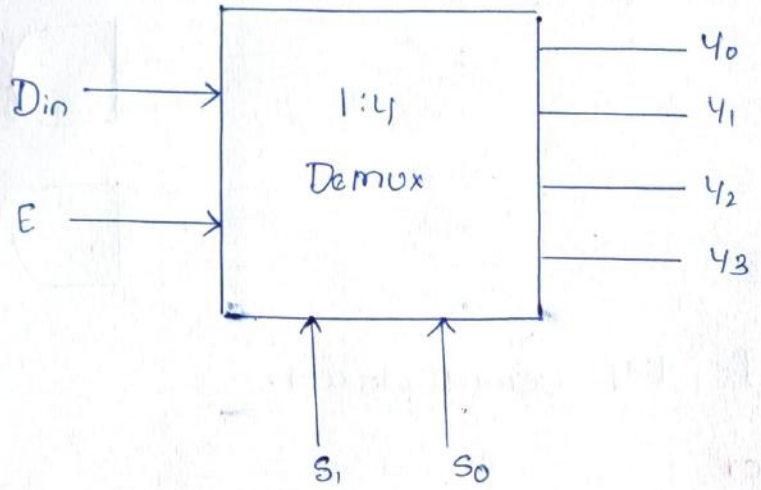


fig:- Block diagram of 1:4 Demux

Enable	Data Input	Select lines		outputs			
E	Din	S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	x	x	x	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

Truth table of 1:4 Demux

$Y_0 = E\overline{S_1}\,\overline{S_0}\,D_{in}$

$Y_1 = E\,\overline{S_1}\,S_0\,D_{in}$

$Y_2 = E\,S_1\,\overline{S_0}\,D_{in}$

$Y_3 = E\,S_1\,S_0\,D_{in}$

$Y_0 = E\,\bar{S_2}\,\bar{S_1}\,\bar{S_0}\,D_{in}$

$Y_1 = E\,\bar{S_2}\,\bar{S_1}\,S_0\,D_{in}$

$Y_2 = E\,\bar{S_2}\,S_1\,\bar{S_0}\,D_{in}$

$Y_3 = E\,\bar{S_2}\,S_1\,S_0\,D_{in}$

$Y_4 = E\,S_2\,\bar{S_1}\,\bar{S_0}\,D_{in}$

$Y_5 = E\,S_2\,\bar{S_1}\,S_0\,D_{in}$

$Y_6 = E\,S_2\,S_1\,\bar{S_0}\,D_{in}$

$Y_7 = E\,S_2\,S_1\,S_0\,D_{in}$

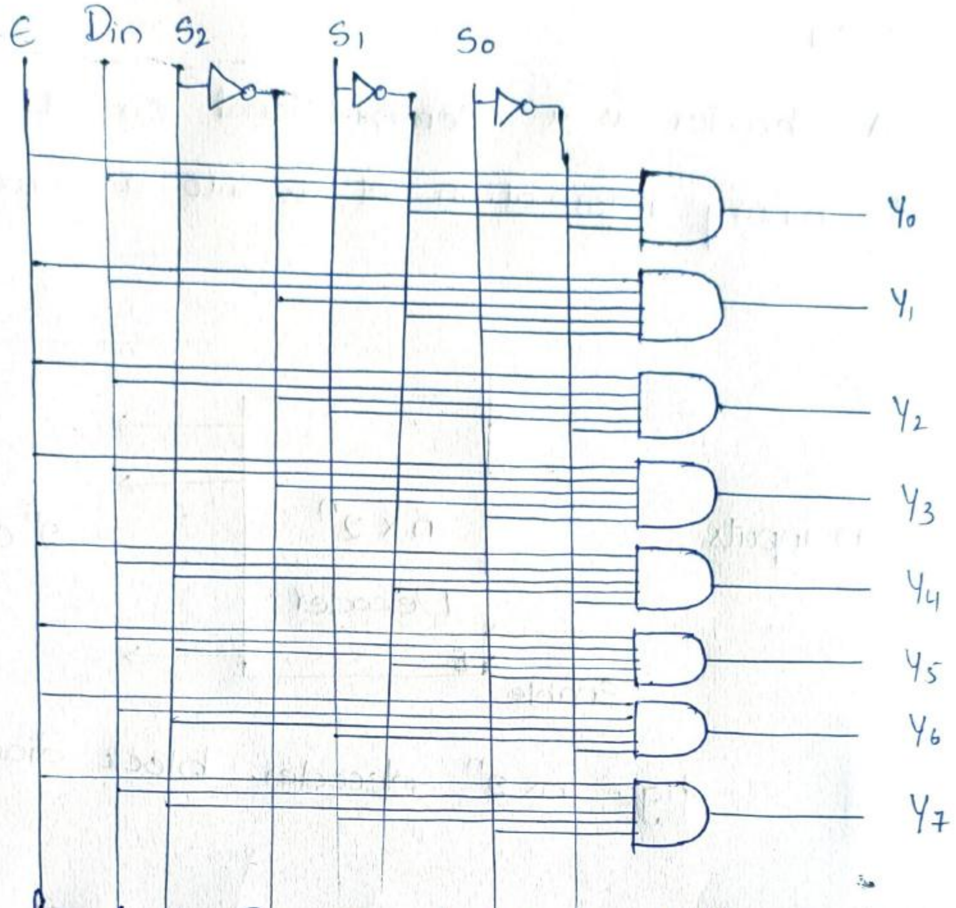


fig:- Logic Circuit of 1:8 demultiplexer

Obtain a 1:4 line demux using 1:2 demultiplexers.

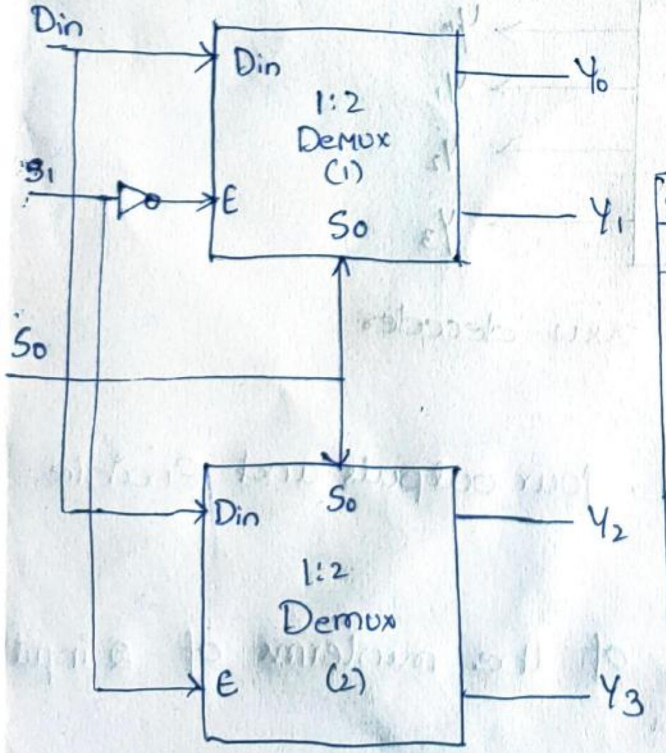


fig:- 1:4 Demux using 1:2 Demux

Select Inputs		Outputs				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	
0	0	0	0	0	D_{in}	Demux (1) Enabled
0	1	0	0	D_{in}	0	
1	0	0	D_{in}	0	0	Demux (2) Enabled
1	1	D_{in}	0	0	0	

Decoder:-

A decoder is a combinational circuit. It converts the n-bit binary information at its into a maximum of 2^n output lines.

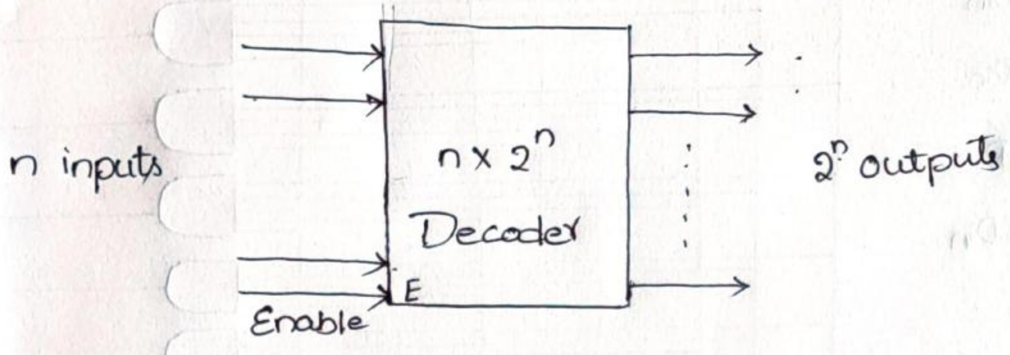


fig:- $n \times 2^n$ decoder block diagram

2×4 Decoder:-

The block diagram of 2×4 decoder has shown in figure.

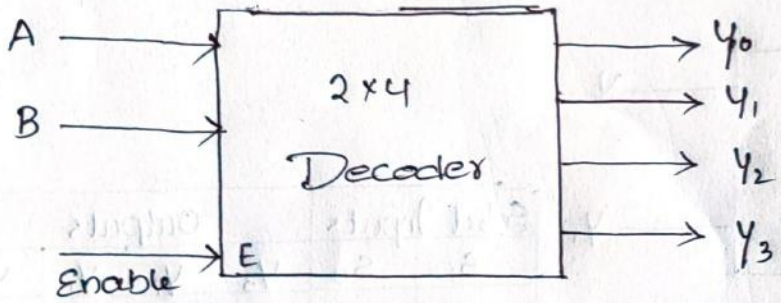


fig:- Block diagram of 2×4 decoder

- It ~~consists~~ Contains 2 inputs, four outputs and Enable input.
- Each output represents one of the minterms of 2 input variables
- If Enable =0, then all the outputs are zero.
- If Enable =1, one of the outputs Y_0 to Y_3 is active

for a given input.

Inputs			Outputs			
E	A	B	Y_0	Y_1	Y_2	Y_3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

output Y_0 is active, $Y_0=1$ when inputs $A=0, B=0$

Y_1 is Active, $Y_1=1$ when inputs $A=0, B=1$

Y_2 is Active, $Y_2=1$ when inputs $A=1, B=0$

Y_3 is Active, $Y_3=1$ when inputs $A=1, B=1$

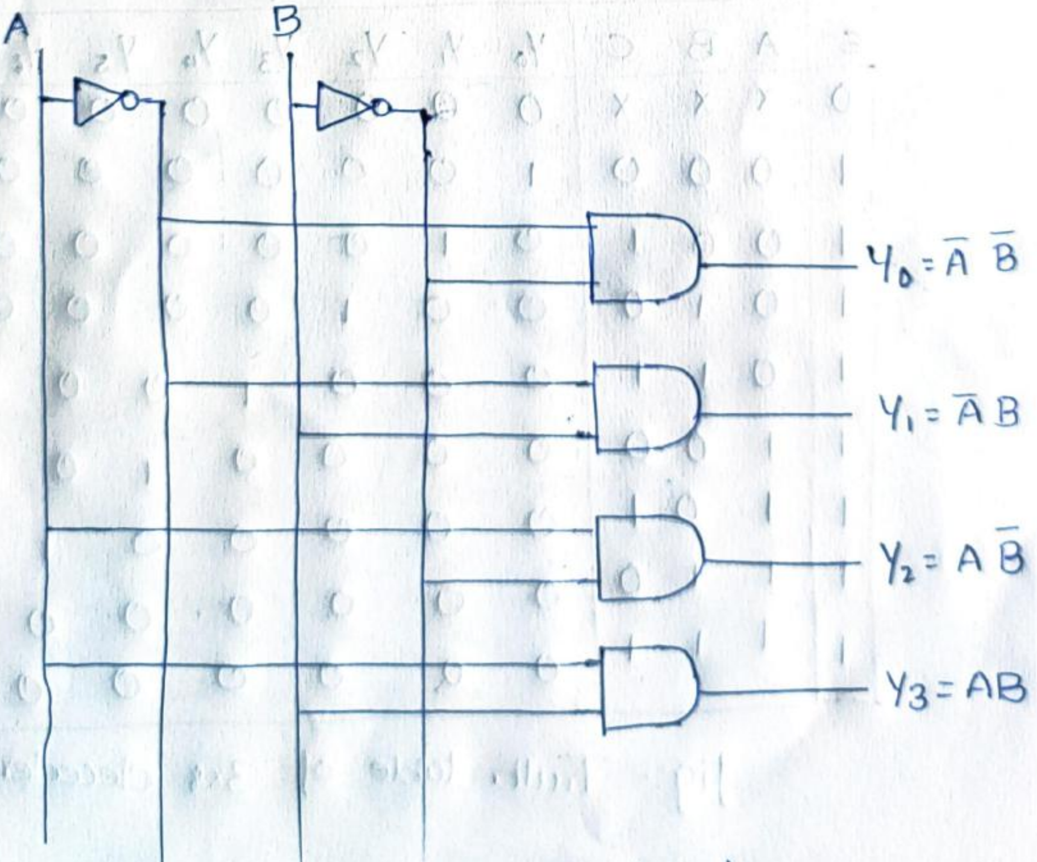


fig 1 - Logic Diagram of 2x4 decoder

3x8 Decoder:-

The block diagram of 3x8 decoder is shown in figure.

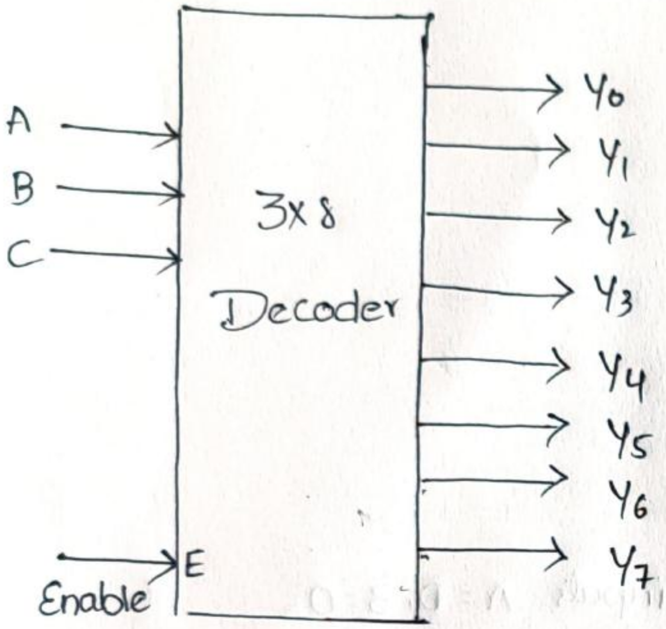


fig:- Block diagram of 3x8 decoder.

Inputs				Outputs							
E	A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

fig:- Truth table of 3x8 decoder

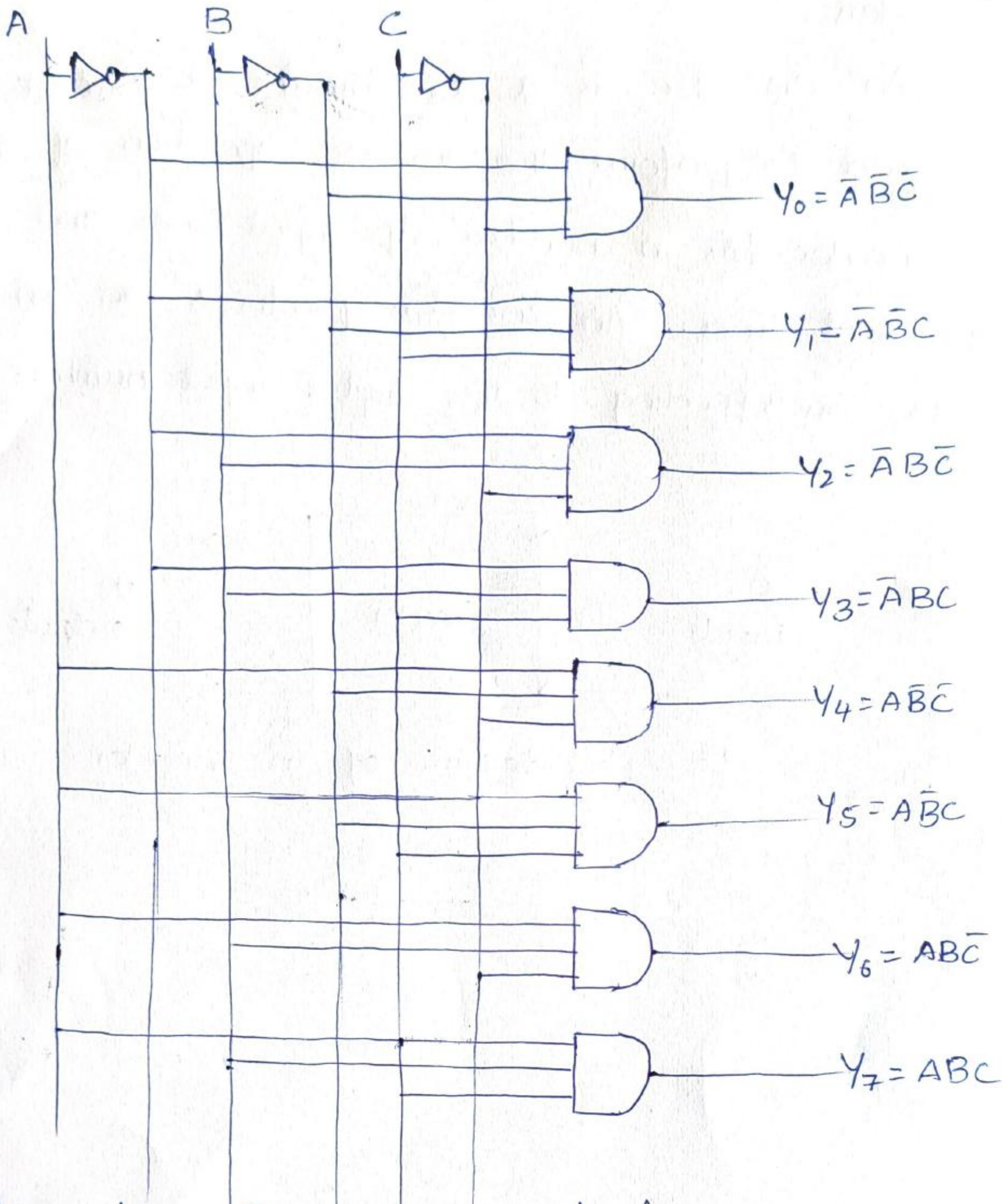


fig:- Logic Diagram of 3x8 decoder

Applications of Decoders:-

- They can be used as code converters

Ex:- BCD to 7-Segment display decoder

Binary to octal decoder

- They are used for data distribution
- Used as building blocks in implementing Switching function.

UNIT-3

Sequential Circuits :-

The output of a sequential circuit depends upon the present time inputs, the previous output and the sequence in which the inputs are applied.

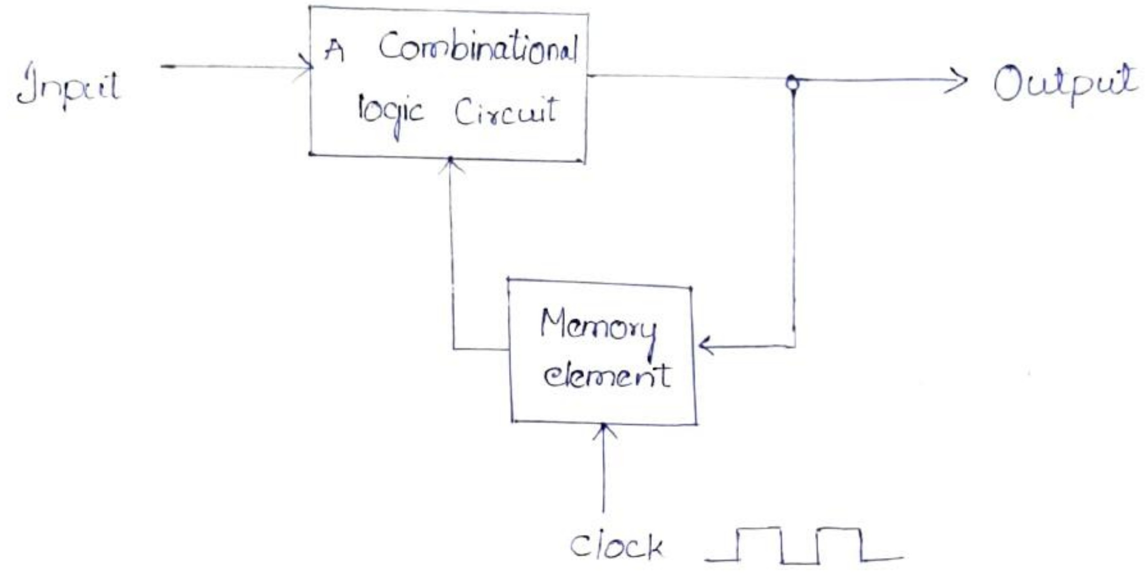


fig:- Block diagram of a sequential circuit.

a sequential circuit requires a memory element.

Present State of a Sequential Circuit:-

The data stored by the memory element at any given instant of time is known as the present state of the sequential circuit.

Next State:-

The Combinational circuit operates on the external inputs and the present state to produce new outputs Some of those new outputs are stored in memory element and known as the next state of the sequential circuit.

Clock Signal:-

The clock signal is a timing signal. clock is a rectangular signal as shown in figure with a duty cycle equal to 50 %. The clock signal repeats itself after every T seconds. Therefore, the clock frequency

$F = 1/T$.

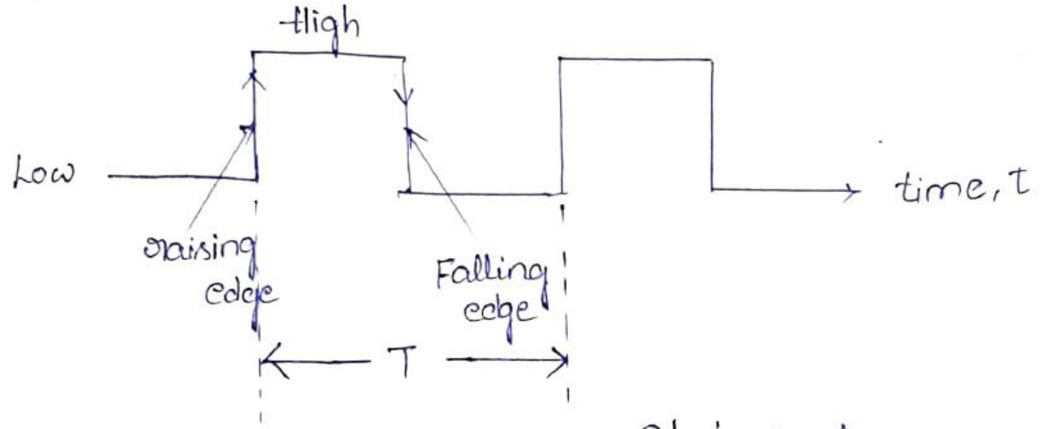


fig:- Illustration of a Clock signal.

Comparison between combinational and sequential circuits:-

Combinational Circuit	Sequential Circuit
* In combinational circuits, the output variables at any instant of time are dependent only on the present input variables.	* In sequential circuits the output variables at any instant of time are dependent not only on the present input variables, but also on the present state.
* Memory unit is not requires in combinational circuit.	* Memory unit is required to store the past history of the input variables.
* These circuits are faster because the delay between the i/p and o/p due to propagation delay of gates only.	* Sequential circuits are slower than Combinational circuits.
* Easy to design.	* Comparatively hard to design.

SR LATCH:-

The latch has two outputs Q and Q'. when the Circuit is switched on the latch may enter into any state. If Q=1, then Q'=0, which is called SET state. If Q=0, then Q'=1, which is called ~~same~~ stav RESET state. Whether the latch is in SET state or RESET state, it will continue to remain in the same state, as long as the power is not switched off. But the latch is not an useful circuit, since there is no way of entering the desired input.

NOR Latch:.

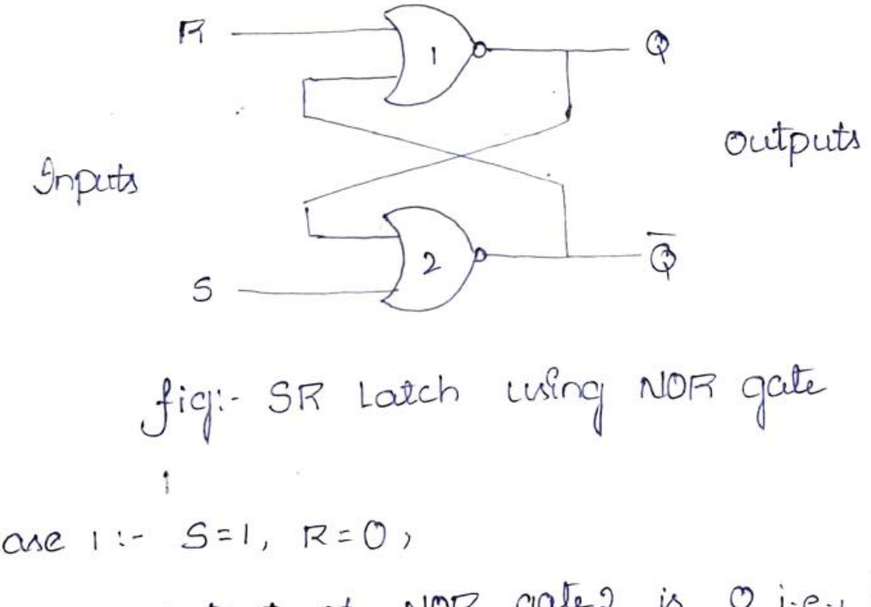


fig:- SR Latch using NOR gate

NOR Truth Table

A	B	$Y=\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Case 1:- S=1, R=0,

output of NOR gate 2 is 0 i.e., $\bar{Q}=0$.

hence the inputs of NOR gate 1 are $0 \Rightarrow Q=1$.

$\therefore$ When $S=1, R=0 \Rightarrow Q=1, \bar{Q}=0$.

Case 2:- S=0, R=1,

Output of NOR gate 1 is 0 because one of the input is 1 in NOR truth table then the output is zero.

$\therefore \bar{Q}=0$

hence the both inputs of NOR gate 1 are 0 so the output $Q = 1$.

$\therefore\ S = 0,\ R = 1 \Rightarrow Q = 1,\ \bar{Q} = 0$

Case 3:- $S = 0,\ R = 0$

we know that $\bar{Q} = \overline{S + Q}$ and $Q = \overline{R + \bar{Q}}$

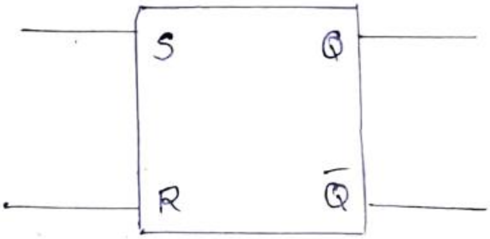
$$\therefore\ \bar{Q} = \overline{0 + Q} \quad \text{and} \quad Q = \overline{0 + \bar{Q}}$$

$$\bar{Q} = \bar{0} \cdot \bar{Q} \qquad Q = \bar{0} \cdot Q$$

$$= \bar{Q} \qquad = Q$$

$\therefore\ S = R = 0$, then Q and $\bar{Q}$ do not change their states.

Case 4:- $S = 1,\ R = 1$, then the outputs Q and $\bar{Q}$ both are forced to 0. Actually, this is an indeterminate state which must be avoided.



Symbol

S	R	Q	$\bar{Q}$	State
0	0	Q^+	$\bar{Q}^+$	Storage state
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Invalid

Truth table

NAND LATCH

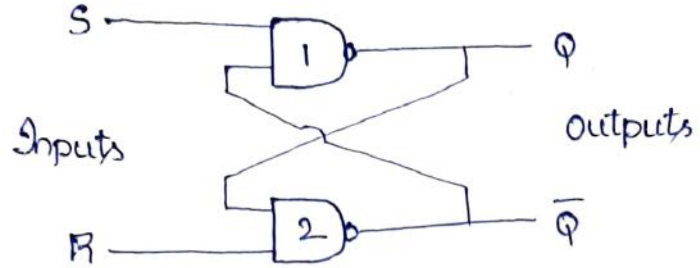


fig:- SR Latch using NAND gates

NAND truth table

A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

case (i):- S=0, R=0; Then the outputs Q and $\overline{Q}$ both are forced to 1. This is an undeterminate state which must be avoided.

case (ii):- S=0, R=1

output of NAND gate 1 is 1 i.e., Q=1

hence the inputs of NAND gate2 are 1 ∴ $\overline{Q}=0$.

case (iii):- S=1, R=0

Output of NAND gate 2 is 1 i.e., $\overline{Q}=1$

hence the inputs of NAND gate1 are 1 ∴ Q=0.

case (iv):- S=1, R=1

we know that $Q = \overline{S \cdot \overline{Q}}$ and $\overline{Q} = \overline{R \cdot Q}$

$$
\begin{aligned}
Q &= \overline{\overline{Q} \cdot S} & \qquad \overline{Q} &= \overline{R \cdot Q} \\
&= \overline{S} + Q & &= \overline{R} + \overline{Q} \\
&= 0 + Q & &= 0 + \overline{Q} \\
&= Q & &= \overline{Q}
\end{aligned}
$$

∴ S=R=1, then Q and $\overline{Q}$ do not change their states.

S	R	Q	$\bar{Q}$	State
0	0	1	1	Invalid
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q^+	$\bar{Q}^+$	Memory

S-R Flip-flop

The basic flip-flop is a one bit memory cell that gives the fundamental idea of memory device. The logic diagram and the block diagram of S-R flip-flop with clocked input.

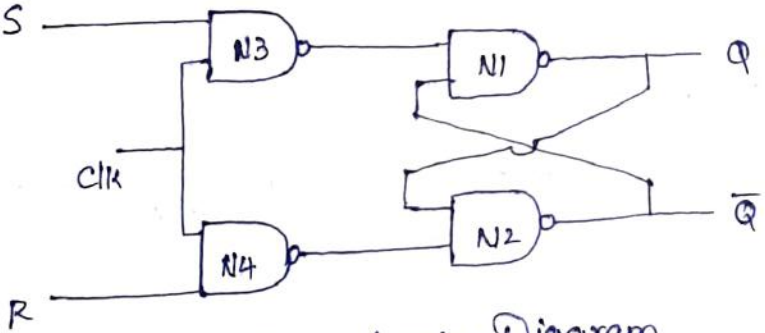


fig:- Logic Diagram

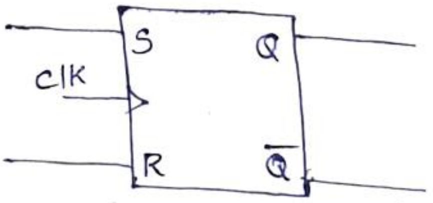


fig:- Block diagram

The flip-flop can be made to respond only during the occurance of clock pulse by adding two NAND gates to the input latch. So synchronization is achieved. i.e., flipflops are allowed to change their states only at particular instant of time. The clock pulses are generated by a clock pulse generator. The flip-flops are affected only with the arrival of clock pulse.

→ When Clk=0 the output of N3 and N4 are 1 regardless of the value of S and R. This is given as input to N1 and N2. This makes the previous value of Q and $\bar{Q}$ unchanged.

→ When Clk=1 the information at S and R inputs are allowed to reach the latch and change of state in flipflop takes place.

→ Clk=1, S=0, R=1 gives the RESET state i.e., Q=0, $\bar{Q}$=1

→ Clk=1, S=1, R=0 gives the SET state i.e., Q=1, $\bar{Q}$=0

→ Clk=1, S=1, R=1 is not allowed, because it is not able to determine the next state. This condition is said to be a "Race Condition".

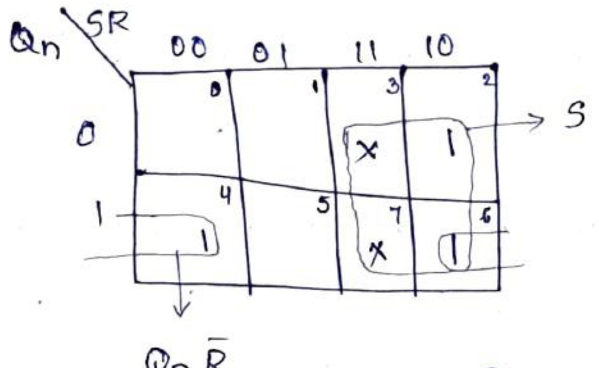
Clk	S	R	Q_{n+1}	
0	x	x	Q_n	Nochange
1	0	0	Q_n	No change
1	0	1	0	RESET
1	1	0	1	SET
1	1	1	x	Not allowed.

Truth table

Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	x
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	x

Characteristic table

from characteristic table



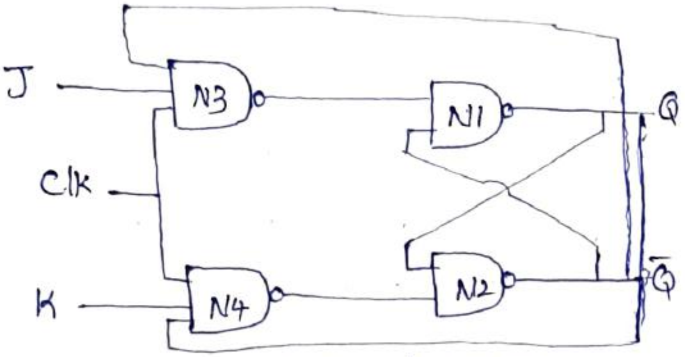
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Excitation table

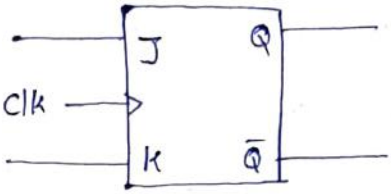
$\Rightarrow Q_{n+1} = S + Q_n\bar{R}$

JK flip-flop

The Invalid condition in SR flip-flop, when S=R=1 is eliminated in J-K flipflop. There is a feedback from the output to the inputs.



Logic diagram



Block diagram

The J and K are called control inputs', because they determine what the flipflop does when a positive clock arrives.

→ When J=0, K=0 then both N3 and N4 will produce high output and the previous value of Q and $\bar{Q}$ retained as it is.

→ When J=0, K=1, N3 will get an output as 1 and o/p of N4 depends on the value of Q. The final output is Q=0, $\bar{Q}$=1 i.e., RESET state.

→ when J=1, K=0 the output of N4 is 1 and N3 depends on the value of $\bar{Q}$. The final output is Q=1 and $\bar{Q}$=0 i.e., set state.

→ when J=1, K=1 it is possible to set (or) reset the flip-flop depending on the current state of output. If Q=1, $\bar{Q}$=0 then N4 passes '0' TO N2 which Produces $\bar{Q}$=1, Q=0 which is reset state. when J=1, K=1, Q changes to the complement of the last

State. The flipflop is said to be in the toggle State

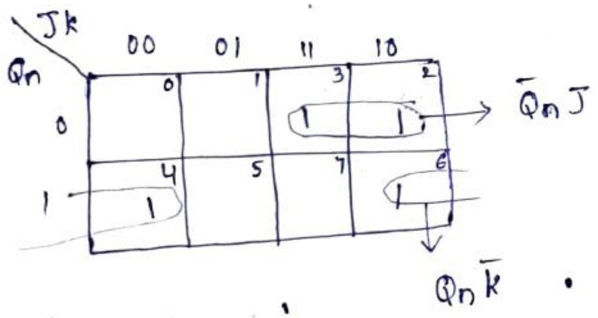
J	K	Q_{n+1}	
0	0	Q_n	Nochange
0	1	0	Reset
1	0	1	Set
1	1	$\overline{Q_n}$	complement/ Toggle

Truth Table

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Characteristic table

from characteristic Table



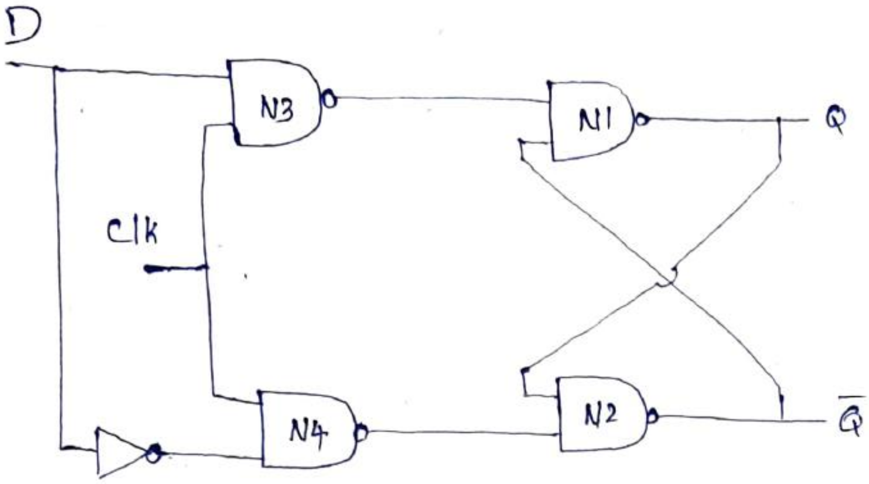
$Q_{n+1} = \overline{Q_n}J + Q_n\overline{k}$

Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

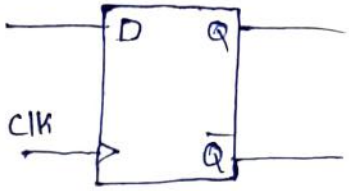
Excitation table

D flip-flop:-

The D flip-flop is the modified form of S-R flipflop, S-R flip-flop is converted to D flip-flop by adding an inverter between S and R and only one input D is taken instead of S and R. So one input is D and complement of D is given as another input. The logic diagram and the block diagram of D flip-flop with clocked input.



Logic diagram



Block diagram

When the clock is low both the NAND gates (N1 and N2) are disabled and Q retains its last value. When clock is high both the gates are enabled and the input value at D is transfered to its output Q. D flipflop is also called "Data flip-flop".

Clk	D	Q_{n+1}
0	x	Q_n
1	0	0
1	1	1

Truth Table

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic Table

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation Table

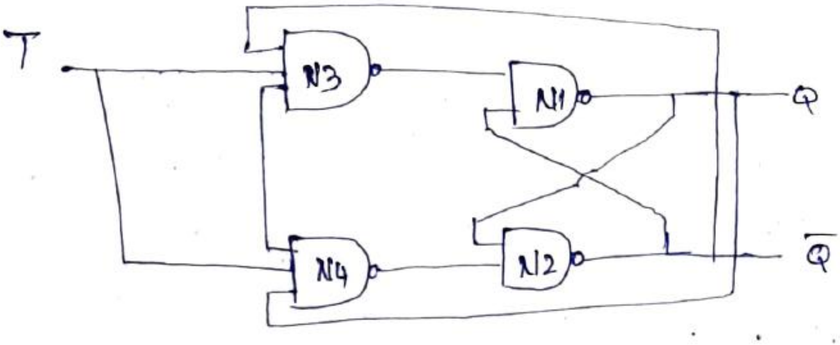
Q_n \ D	0	1
0		1
1		1

D

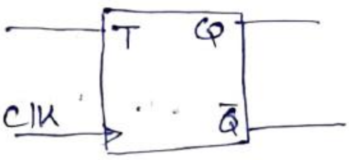
$$\boxed{Q_{n+1} = D}$$

T flipflop:-

If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value.



Logic diagram



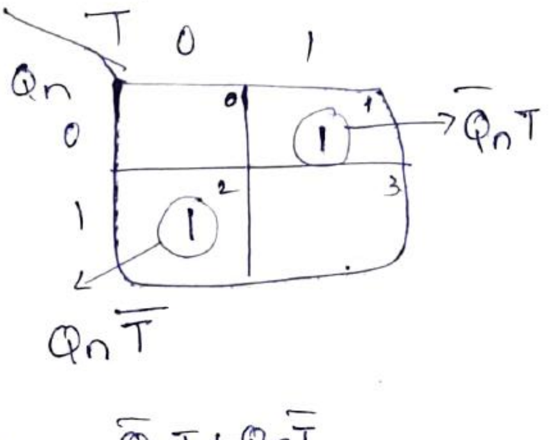
Symbol for T flipflop

Clk	T	Q_{n+1}
0	x	Q_n
1	0	Q_n
1	1	$\overline{Q_n}$

Truth Table

Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic Table



$$Q_{n+1} = \overline{Q}_nT + Q_n\overline{T}$$

Applications of Flip-Flops:-

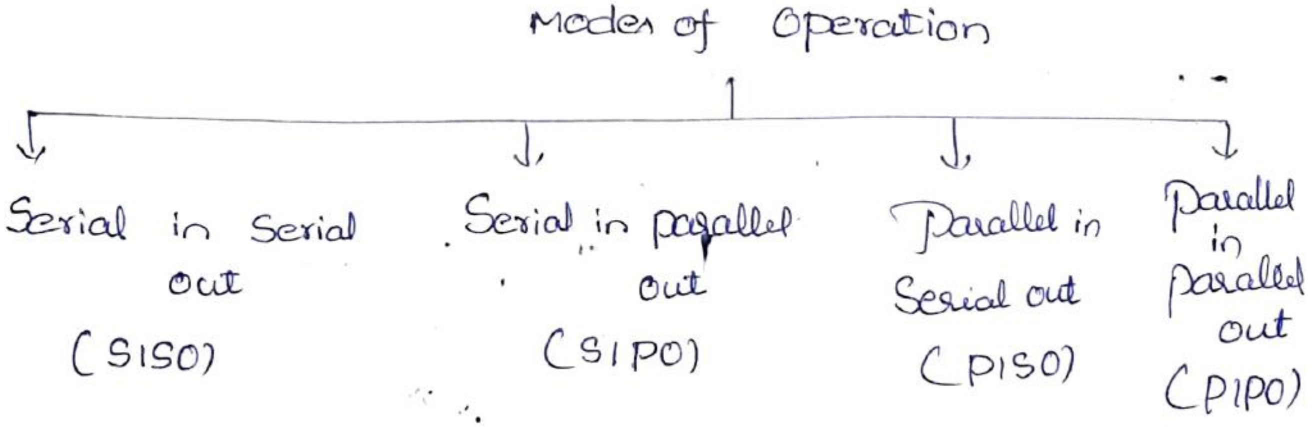
- Frequency Division
- Parallel data storage
- Serial data storage.
- Transfer of data.

Registers

To increase the storage capacity in terms of number of bits, we have to use no. of flipflops. Such a group of flip-flops is known as a Register.

The n-bit register will consist of n number of flipflops and it is capable of storing an n bit word.

Registers may be classified based on the way in which data are entered and taken out from a register. There may be following four possible modes:-



Modes of Operation

Serial in Serial out (SISO)

Serial in parallel out (SIPO)

Parallel in Serial out (PISO)

Parallel in parallel out (PIPO)

Shift Registers:-

The binary data in a register can be moved within the register from one flipflop to the other or outside it with application of clock pulses. Those registers which allow such data transfers are known as Shift Registers.

Modes of operation of a Shift Register.

- Serial input Serial output (serial shift Right)
- Serial input Serial output (Serial shift left)
- Serial input parallel output
- parallel input Serial output.

Serial Input Serial Output (Shift Left Mode)

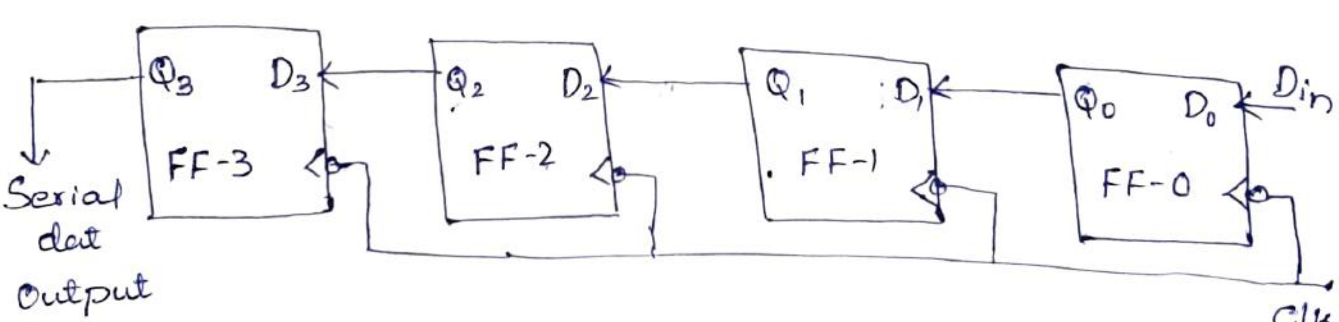


fig:- Serial shift left Register.

→ Let us consider that all the flipflops initially are in the rest condition. i.e., $Q_3 = Q_2 = Q_1 = Q_0 = 0$.

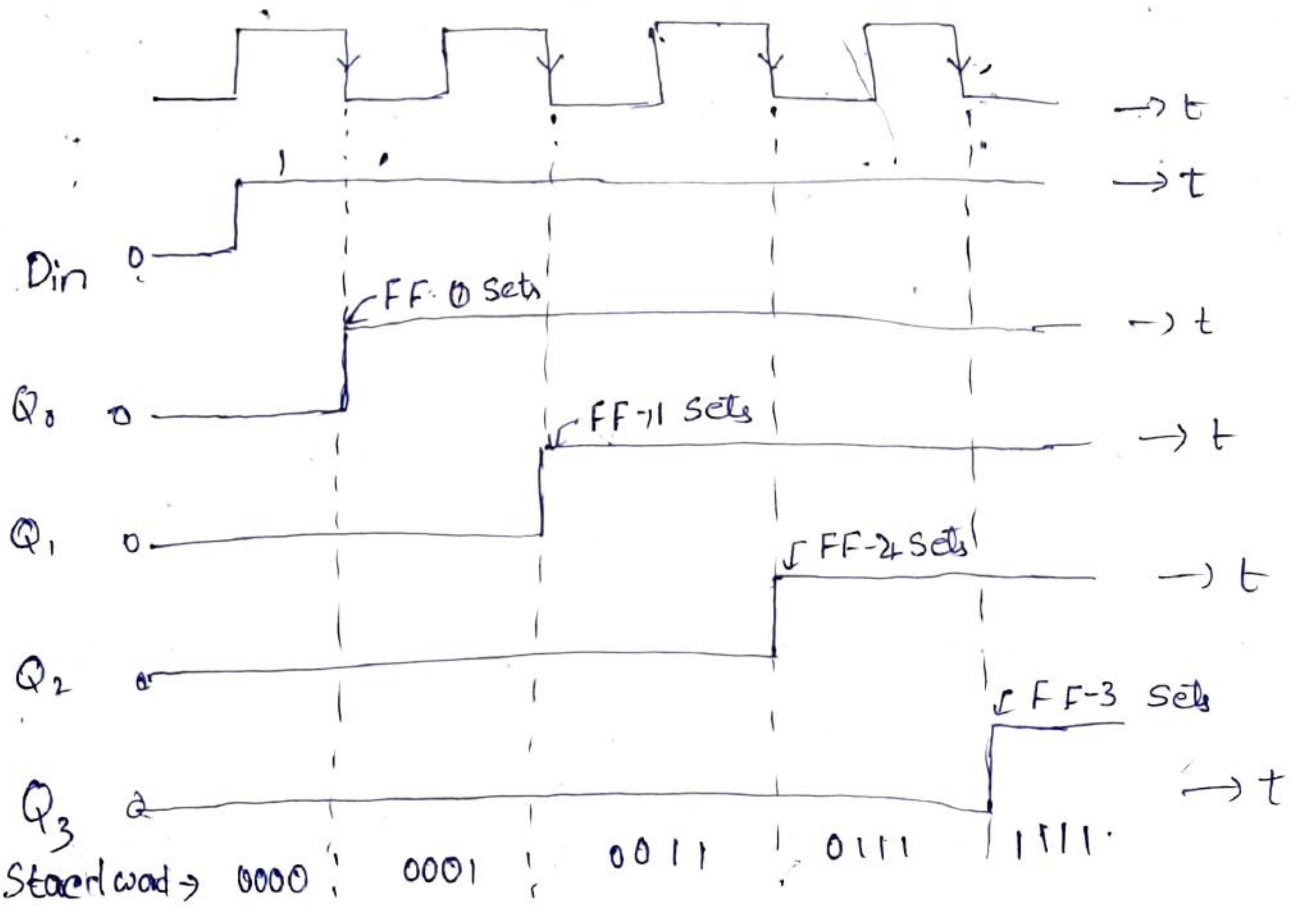
→ Let us illustrate the entry of a four bit binary number 1111 into the register.

This Number must be applied to D_{in} bit-by-bit with the MSB bit applied first.

CLK	Q_3	$Q_2 = D_3$	$Q_1 = D_2$	$Q_0 = D_1$	Serial input $D_{in} = D_0$
	0	0	0	0	
↓	0	0	0	1 ←	1
↓	0	0	1	1 ←	1.
↓	0	1	1	1 ←	1
↓	1	1	1	1 ←	1

Direction of data travel ←

Waveform for Shift Left operation



Serial in Serial Out (shift Right Mode)

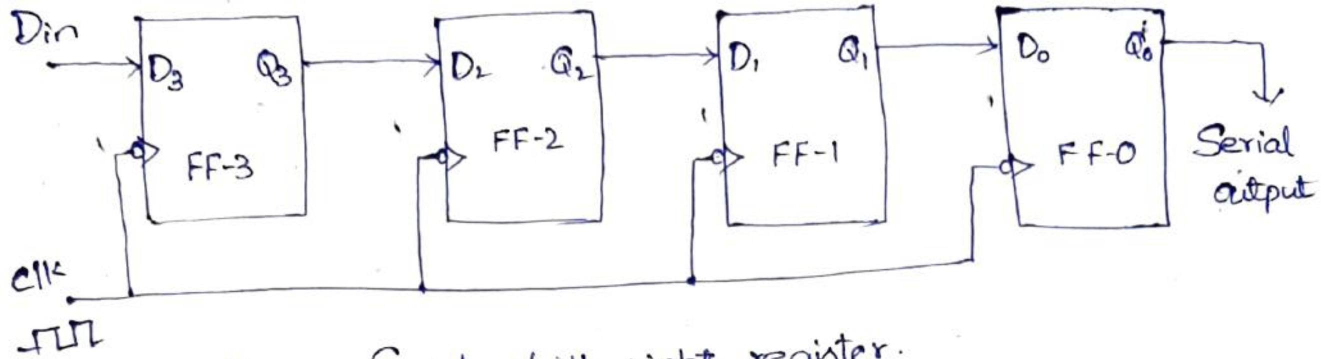


fig:- Serial shift right register.

Before application of clock signal, let Q_3 Q_2 Q_1 Q_0 = 0000 and we apply LSB bit of the number to be entered to Din. Hence Din = D_3 = 1. Then, we apply the clock. On the first falling edge of clock, the FF-3 is set, and the stored word in the register will be given by

$$Q_3\ Q_2\ Q_1\ Q_0 = 1\ 0\ 0\ 0$$

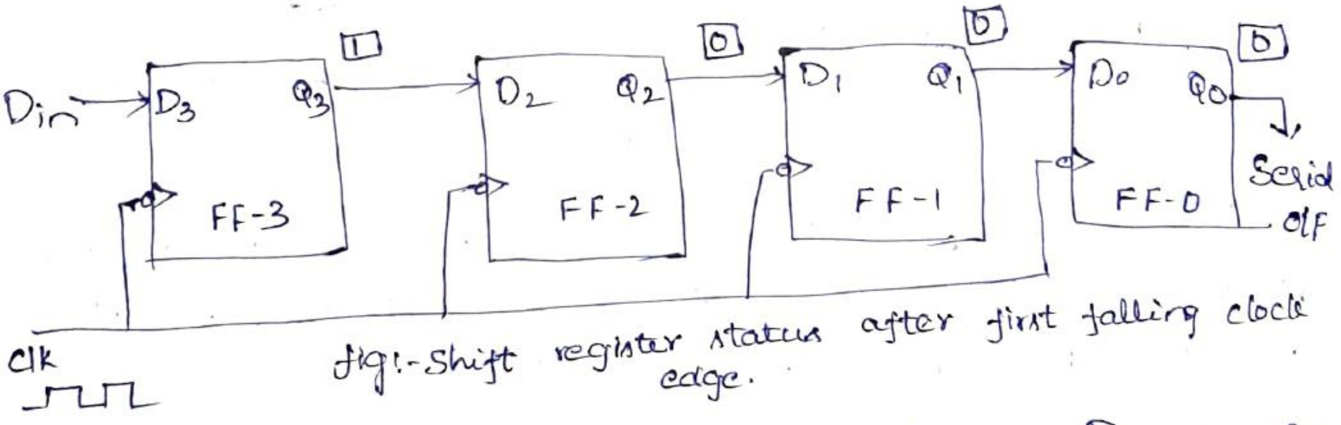


fig:- Shift register status after first falling clock edge.

Apply the next bit to Din. Hence, Din = 1. As soon as the next negative edge of the clock hits, FF-2 sets and the stored word will change to, $Q_3 \; Q_2 \; Q_1 \; Q_0 = 1 \; 1 \; 0 \; 0$

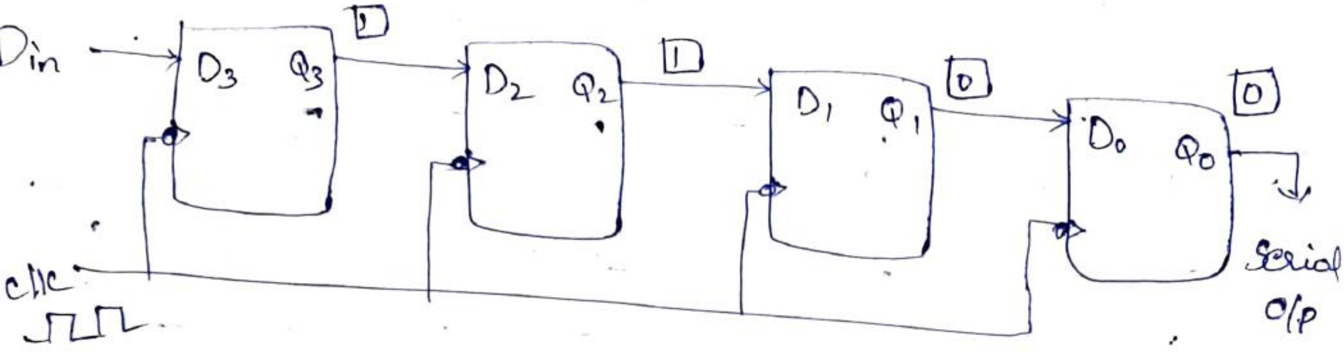


fig:- Shift register status after second falling edge of clock.

Apply next bit to be stored i.e., 1 to Din. Then apply the clock pulse. As soon as the third negative clock edge hits. FF-1 will sets and the Output will get modified to

$Q_3 \; Q_2 \; Q_1 \; Q_0 = 1 \; 1 \; 1 \; 0$

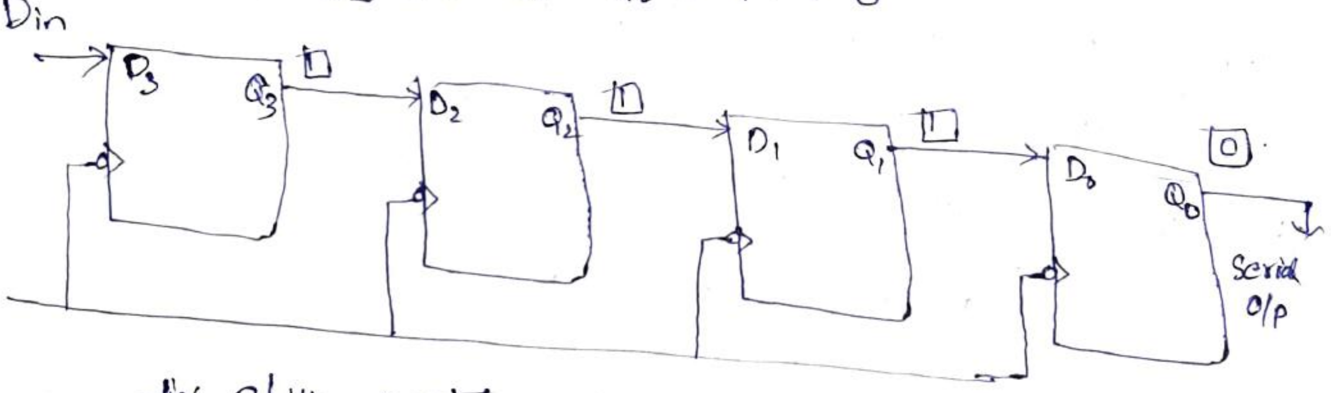


fig:- Shift register status after the third falling edge of clock.

Similarly with Din = 1 with the fourth negative clock edge arriving, the stored word in the register will be given by $Q_3 \; Q_2 \; Q_1 \; Q_0 = 1111$

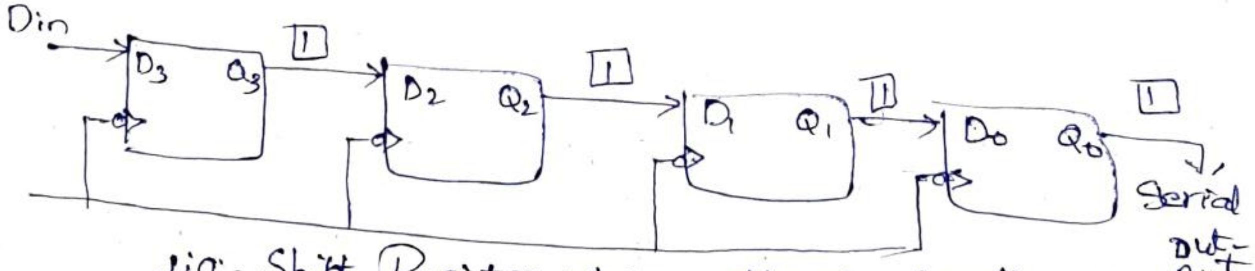
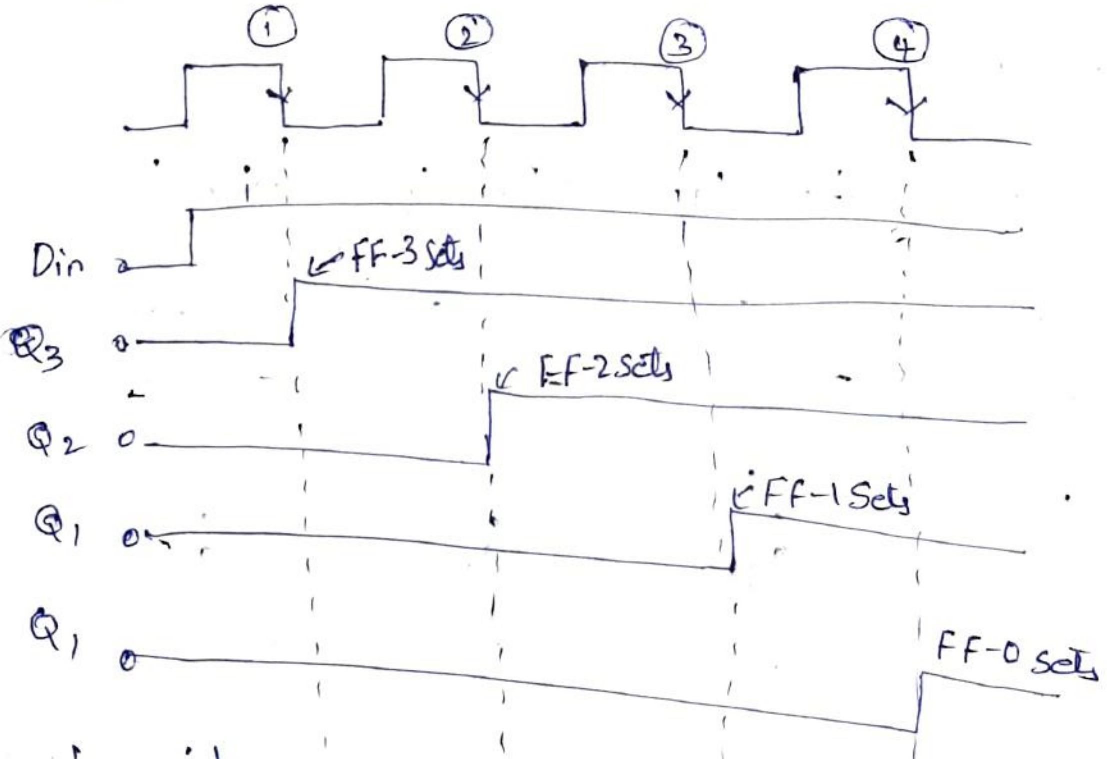


fig:- Shift Register status after the fourth falling edge of clock.

CLK	Din = Q3	Q3 = D2	Q2=D1	Q1= D0	Q0=D1
		0	0	0	0
↓	1 →	1	0	0	0
↓	1 →	1	1	0	0
↓	1 →	1	1	1	0
↓	1 →	1	1	1	1

→ Direction of data travel.

Waveforms for Shift Right Operation



stored words 0000 1000 1100 1110 1111.

Serial IN Parallel OUT (SIPO):-

In this data is entered serially and then taken out in parallel. This means that first the data is loaded bit-by-bit. The outputs are disabled as long as the loading is taking place. As soon as the loading is complete, and all the flipflops consist of their required data, the outputs are enabled. So that all the loaded data is made available over all the output lines simultaneously. Also, number of clock cycles required to load a four bit word is 4. Therefore, the speed of operation of SIPO mode will remain same as that of SISO mode.

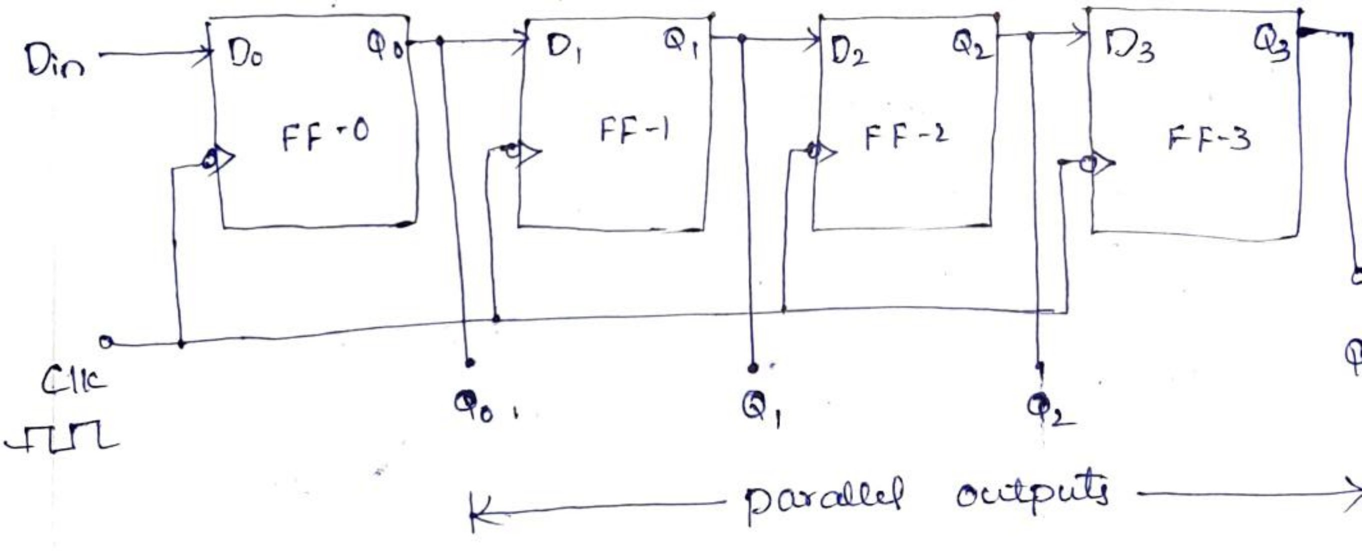
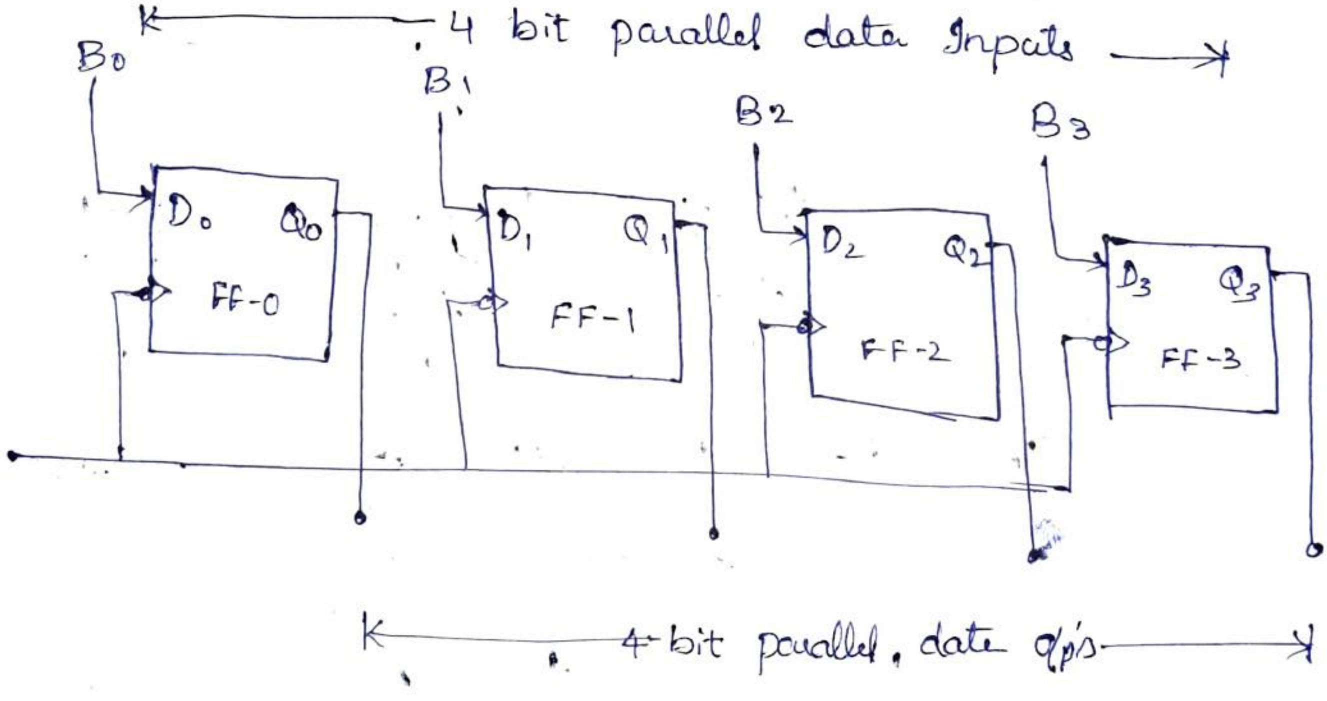


fig:- Illustration of Serial input parallel output mode.

PARALLEL IN PARALLEL OUT (PIPO)

figure shows the parallel in parallel out mode of operation. The 4-bit binary input B_0, B_1, B_2, B_3 is applied to the data inputs D_0, D_1, D_2 and D_3 respectively of the four flip-flops. As soon as a negative clock edge is applied, the input binary bits shall be loaded into the flip-flops simultaneously. The loaded bits appear simultaneously to the output side. Here, only one clock pulse is essential to load all the bits.



PARALLEL IN SERIAL OUT MODE (PISO)

In this mode, the bits are centered in parallel i.e., The circuit shown in figure is a four bit parallel input serial output register. Output of previous FF is connected to the input of the next one with the help of a combinational circuit. Then the binary input word B0, B1, B2, B3 is applied through the same combinational circuit. There are two modes in which this circuit can work. These modes are shift mode and load mode.

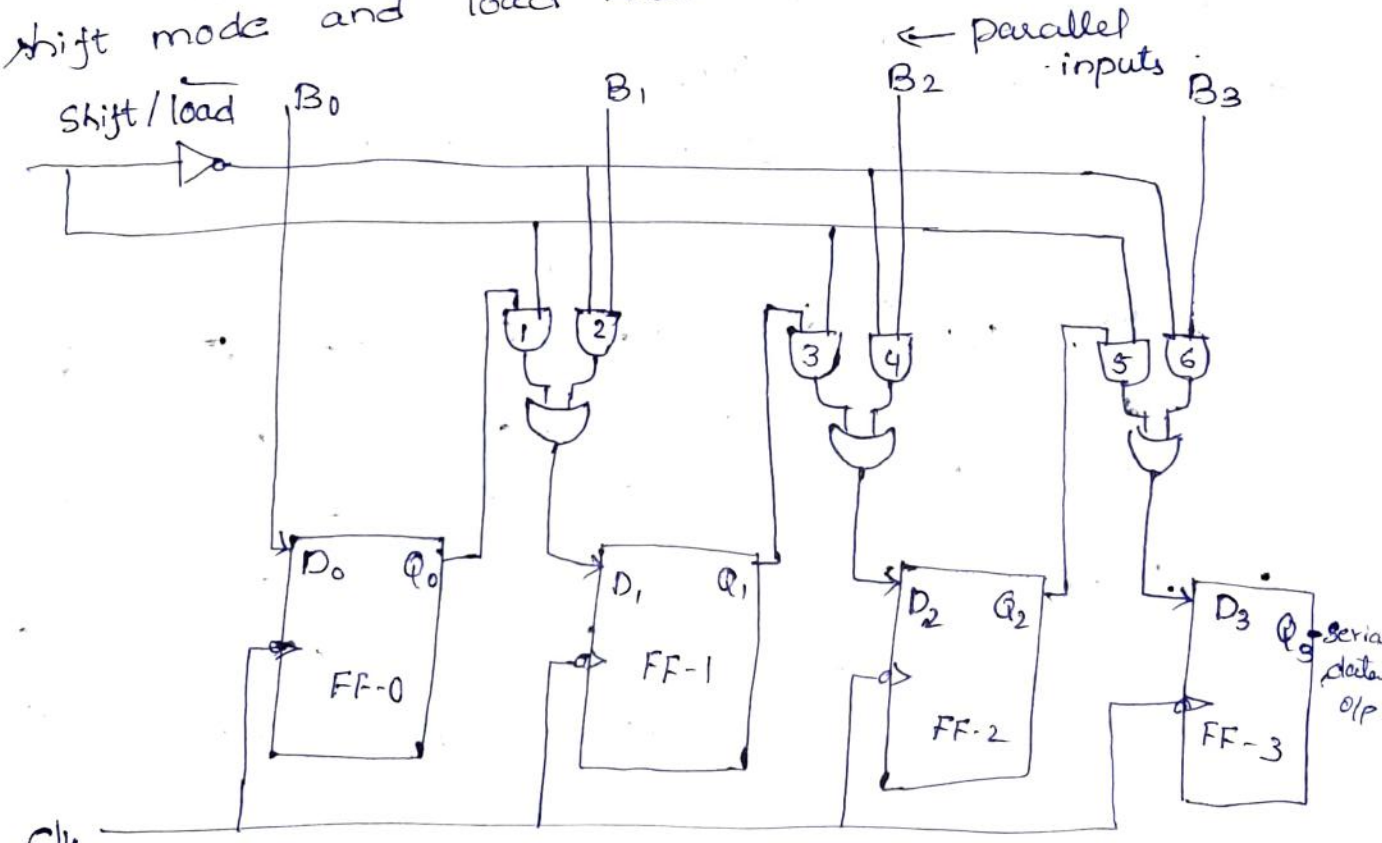


fig:. Parallel in Serial out Shift Register

Shift Mode

When the shift/$\overline{load}$ line is high (1), the AND gates 2, 4, 6 become inactive. Hence, the parallel loading of the data becomes impossible. But, the AND gates 1, 3and 5 become active. Therefore, the shifting of data from left to right bit-by-bit on application of clock pulses. Thus, the parallel in Serial out operation takes place.

Load Mode:

when the shift/$\overline{\text{load}}$ line is low (0), the AND gates 2, 4 and 6 be come active. They pass B_1, B_2 and B_3 bits to the corresponding flip-flops. On low going edge of clock, the binary inputs B_0 $B_1$$B_2$$B_3$ get loaded into the corresponding flip-flops. Therefore, parallel loading takes place.

Counters:-

The digital circuit used for counting pulses is known as counter.

It is a Sequential circuit. counter is the widest applications of flip-flops. It is a group of flip-flops with a clock signal applied. Basically counters count the number of clock pulses. It can be used for measuring frequency or time period.

Classification of Counters

counters are basically of following two types:

i, Asynchronous or ripple counters.

ii, Synchronous counters.

(i) Asynchronous/Ripple up Counter

for these counters, the external clock signal is applied to one flip-flop and then the output of preceding flip-flop is connected to the clock of next flip-flop.

figure shows a 2 bit Ripple up Counter. The no. of flip-flops used is 2. The Toggle flip-flops are being used. But, we can use the JK flip-flop.

Here T is connected permanently to logic 1. External clock is applied to the clock input of flipflop A and Q_A output is applied to the clock input of the next flip-flop i.e., FF-B.

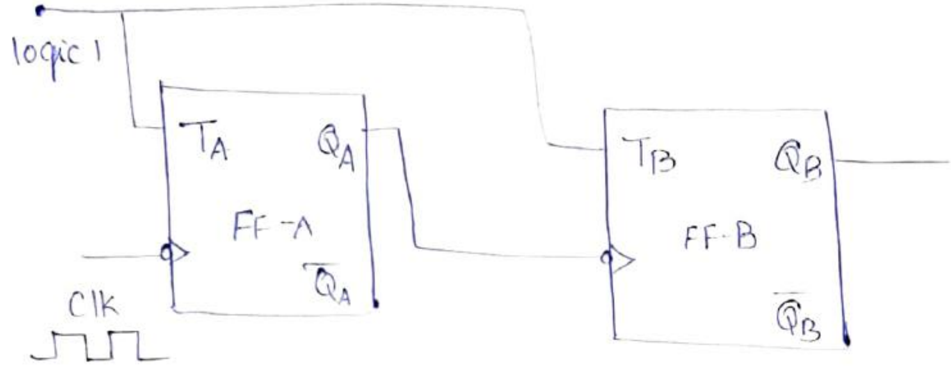


fig:- A two bit asynchronous binary up counter.

→ Initially both the flipflops be in reset condition

Therefore, $Q_B \; Q_A = 0 \, 0$

→ The first negative going clock edge hits FF-A, it will toggle as $T_A = 1$. Hence, Q_A will be equal to 1. Also, Q_A is connected to clock input of FF-B. Since Q_A has changed from 0 to 1, it is treated as the positive clock edge by FF-B. There is no change in Q_B because FF-B is a negative edge triggered FF. Hence after the first clock pulse the counter outputs are $Q_B \; Q_A = 0 \, 1$

→ At the Second falling edge of clock.

FF-A toggles again, to make $Q_A = 0$.

∴ Q_B will become 1.

The counter outputs are

$$Q_B\ Q_A = 1\,0$$

→ At the ~~fourth~~ Third falling edge of clock

FF-A toggles and Q_A Changes from 0 to 1.

∴ Q_B will remains same

The counter outputs are

$$Q_B\ Q_A = 1\,1$$

→ At the fourth falling edge of clock

FF-A toggles and Q_A becomes 0 and Q_B also toggles from 1 to 0.

∴ The counter outputs are

$$Q_B\ Q_A = 0\,0.$$

Clk	Counter outputs		State number	Decimal equivalent of counter o/p
	QB(MSB)	QA(LSB)		
Initially	0	0	—	0
1st (↓)	0	1	1	1
2nd (↓)	1	0	2	2
3rd (↓)	1	1	3	3
4th (↓)	0	0	4	0

from the above table, this counter has four distinct states of output namely 00, 01, 10 and 11. In general, the number of states = 2^n, where n is equal to the number of flip-flops.

Down Counter:-

The counters which can count in the downward direction, i.e., from the maximum count to zero are called down counters.

A 3-bit asynchronous down counter has been shown in figure. The clock input is applied directly to FF-A. But, $\overline{Q_A}$ is connected to clock of FF-B, $\overline{Q_B}$ to clock of FF-C and so on.

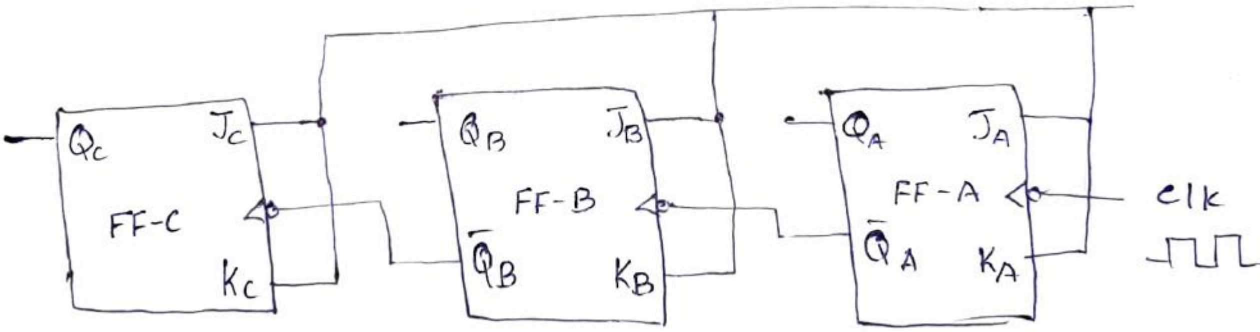


fig:- A 3-bit asynchronous down counter.

Let initially all the flip-flops be in the reset condition.

$\therefore$ $Q_C \; Q_B \; Q_A = 0 \; 0 \; 0$

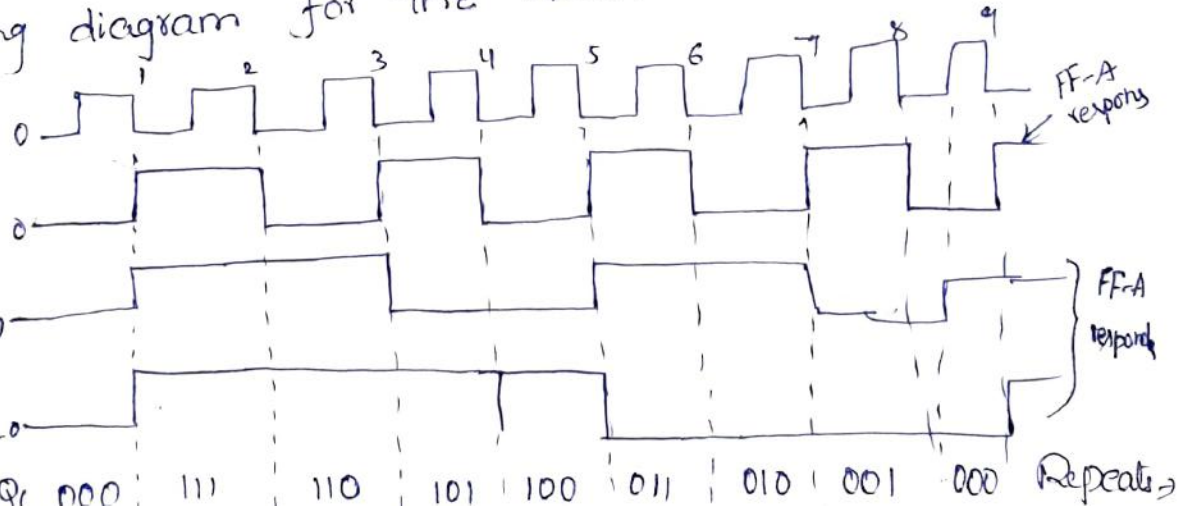
As soon as the first falling clock pulse arrives, FF-A toggles. so, Q_A becomes 1 and $\overline{Q_A}$ becomes 0 from 1. Also, $\overline{Q_A}$ acts as a clock to FF-B. Hence, FF-B will change its state. Hence, Q_B becomes 1 and $\overline{Q_B}$ becomes 0 from 1. $\overline{Q_B}$ acts as clock to FF-C. Hence FF-C will change its state. therefore Q_C becomes 1 and $\overline{Q_B}$ becomes 0. Hence, after the first clock pulse, the output of counter are,

$$Q_C\ Q_B\ Q_A = 111$$

Corresponding to the second falling clock edge, FF-A toggles, Q_A becomes 0 and $\overline{Q_A}$ becomes 1. This positive going change in $\overline{Q_A}$ does not alter the state of FF-B. SO, Q_B remains 1 and $\overline{Q_B}$ remains 0. Hence, there is no change in the state of FF-C. Hence, after the second clock pulse, the counter outputs as under:

$$Q_C\ Q_B\ Q_A = 110$$

The down counting will thus take place. The timing diagram for the down counter is shown.



Synchronous Counters

If the clock pulses are applied to all the flipflops in a counter simultaneously, then such a counter is called as Synchronous counter.

2 - Bit Synchronous UP counter

A. 2-bit synchronous counter has shown in fig: The J_A and K_A inputs of FF-A are connected to logic 1. Hence FF-A works as a toggle flip-flop. The J_B and K_B inputs are connected to Q_A. Hence, FF-B toggles if $Q_A = 1$ and there won't be any state change if $Q_A = 0$.

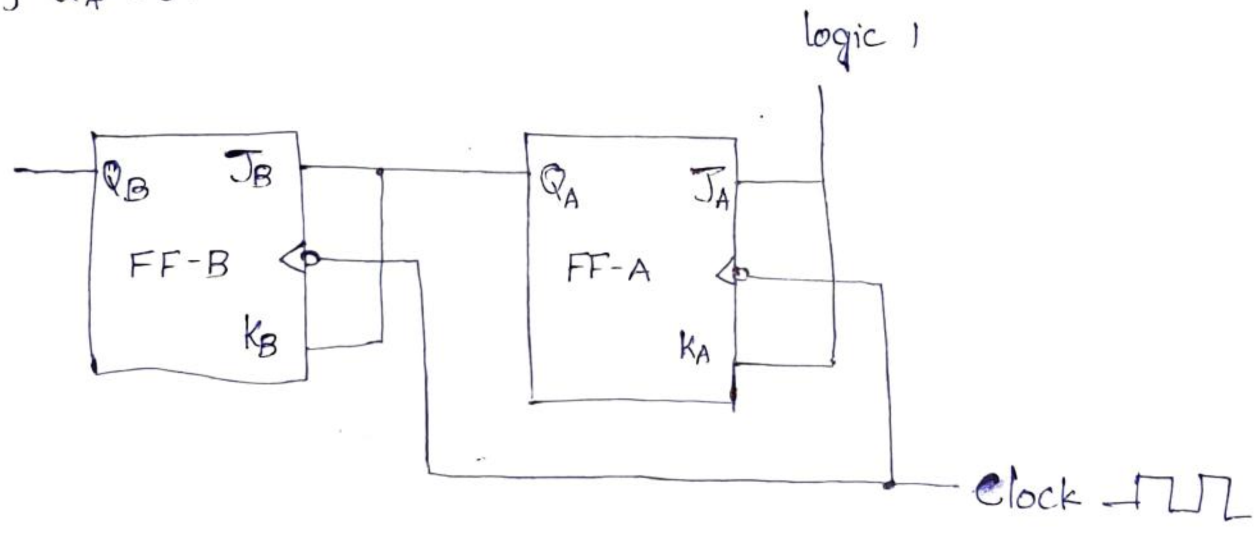


fig:- A 2-bit Synchronous counter.

→ Initially $Q_B\ Q_A = 0\ 0$

→ After first negative clock edge is applied. FF-A toggles and Q_A will changes from 0 to 1. But, at the instant of applications of negative clock edge, $Q_A = 0$, therefore, $J_B = K_B = 0$. Hence FF-B will not change its state. Thus, Q_B will remain 0.

$\therefore Q_B Q_A = 0\ 1$

→ On the arrival of second negative clock edge, FF-A toggles again and Q_A changes from 1 to 0. But, at this instant, Q_A was 1. Hence, $K_B = 1$ and FF-B toggles. Hence, Q_B changes from 0 to 1.

$\therefore Q_B\ Q_A = 1\ 0$

→ Simillarly on the next clock pulse FF-A toggles Q_A from 0 to 1, But there is no change of state for Q_B.

$Q_B\ Q_A = 1\ 1$

→ On the next clock pulse Q_A changes from 1 to 0 as Q_B will also change from 1 to 0.

$Q_B\ Q_A = 0\ 0$.

Clock	Counter outputs	
	Q_B (MSB)	Q_A (LSB)
Initially	0	0
1st (↓)	0	1
2nd (↓)	1	0
3rd (↓)	1	1
4th (↓)	0	0

S.no	parameter of comparison	Asynchronous counter	Synchronous counter
1.	Circuit Complexity	Logic circuit is simple	with increase in no. of states, the logic circuit becomes complicated.
2.	Connection pattern	output of the preceding FF is connected to clock of the next FF.	There is no connection between output of preceding FF and CLK of next one.
3.	Clock input	All the FFs are not Clocked simultaneously.	All the FFs receive clock signal simultaneously.
4.	Propagation Delay	$p \cdot D = n \times (t_d)$ where n is no. of FFs and t_d is p.d per F.F	$p \cdot D = (t_d)_{FF} + (t_d)_{gate}$ It is much shorter than that of asynchronous counter.
5.	Maximum frequency of operation	Low because of the long propagation delay.	High due to shorter propagation delay.

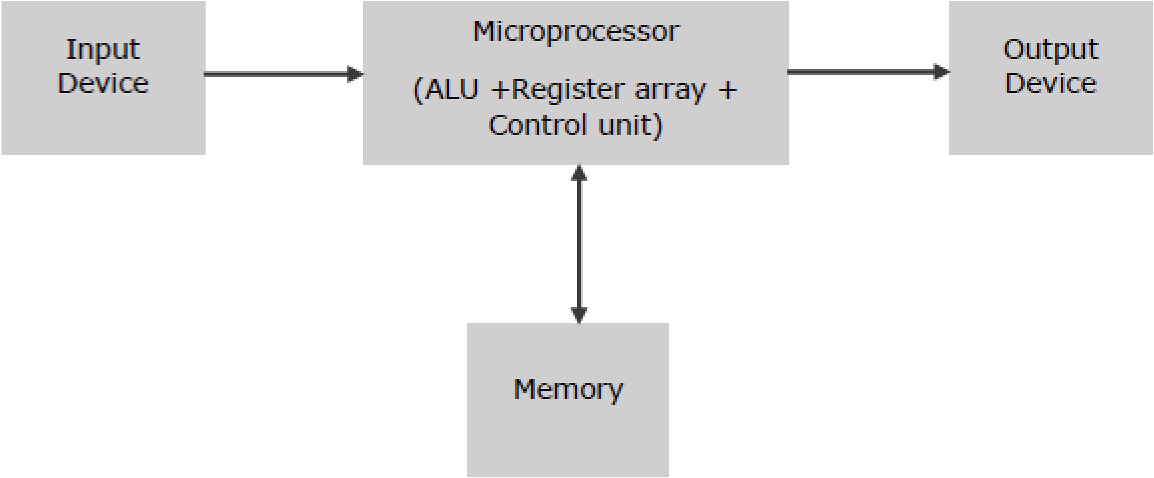
UNIT-IV Microprocessor-I

Microprocessor - Overview

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.

Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer.

Block Diagram of a Basic Microcomputer



How does a Microprocessor Work?

The microprocessor follows a sequence: Fetch, Decode, and then Execute.

Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

List of Terms Used in a Microprocessor

A list of some of the frequently used terms in a microprocessor −

- **Instruction Set** − It is the set of instructions that the microprocessor can understand.
- **Bandwidth** − It is the number of bits processed in a single instruction.

- **Clock Speed** − It determines the number of operations per second the processor can perform. It is expressed in megahertz (MHz) or gigahertz (GHz).It is also known as Clock Rate.
- **Word Length** − It depends upon the width of internal data bus, registers, ALU, etc. An 8-bit microprocessor can process 8-bit data at a time. The word length ranges from 4 bits to 64 bits depending upon the type of the microcomputer.
- **Data Types** − The microprocessor has multiple data type formats like binary, BCD, ASCII, signed and unsigned numbers.

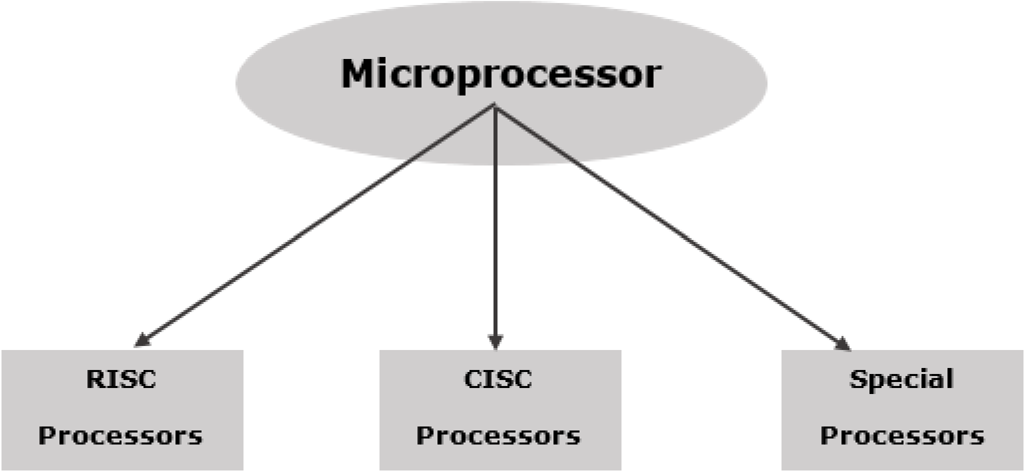
Features of a Microprocessor

features of any microprocessor −

- **Cost-effective** − The microprocessor chips are available at low prices and results its low cost.
- **Size** − The microprocessor is of small size chip, hence is portable.
- **Low Power Consumption** − Microprocessors are manufactured by using metaloxide semiconductor technology, which has low power consumption.
- **Versatility** − The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
- **Reliability** − The failure rate of an IC in microprocessors is very low, hence it is reliable.

Microprocessor - Classification

A microprocessor can be classified into three categories −



RISC Processor

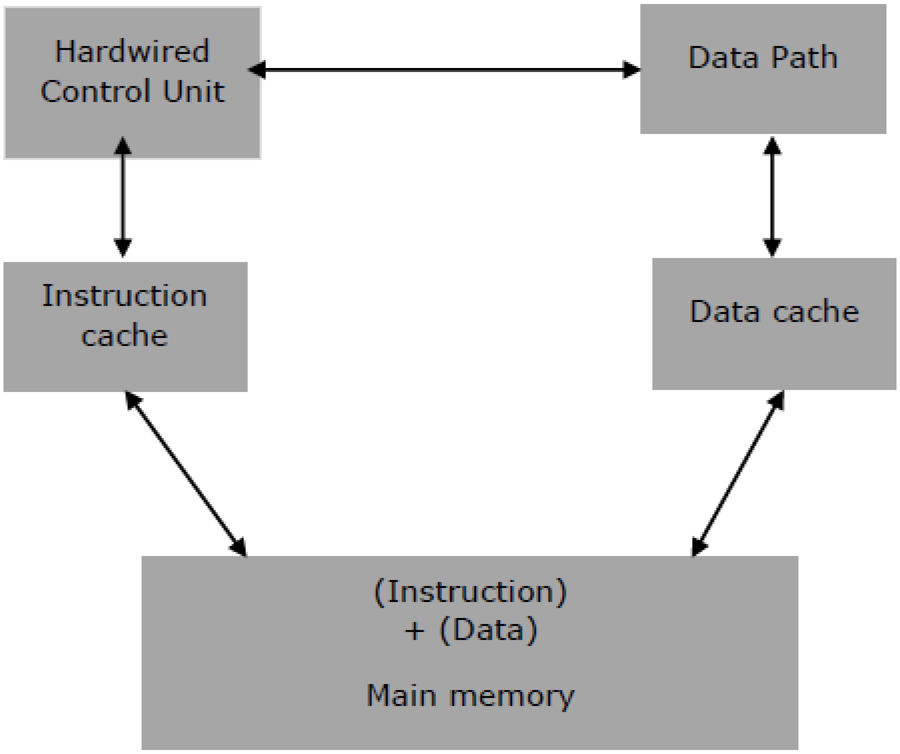
RISC stands for **Reduced Instruction Set Computer**. It is designed to reduce the execution time by simplifying the instruction set of the computer. Using RISC processors, each instruction requires only one clock cycle to execute results in uniform execution time. This reduces the efficiency as there are more lines of code, hence more RAM is needed to store the instructions. The compiler also has to work more to convert high-level language instructions into machine code.

Some of the RISC processors are −

- Power PC: 601, 604, 615, 620
- DEC Alpha: 210642, 211066, 21068, 21164
- MIPS: TS (R10000) RISC Processor
- PA-RISC: HP 7100LC

Architecture of RISC

RISC microprocessor architecture uses highly-optimized set of instructions. It is used in portable devices like Apple iPod due to its power efficiency.



Characteristics of RISC

The major characteristics of a RISC processor are as follows −

- It consists of simple instructions.

- It supports various data-type formats.
- It utilizes simple addressing modes and fixed length instructions for pipelining.
- It supports register to use in any context.
- One cycle execution time.
- “LOAD” and “STORE” instructions are used to access the memory location.
- It consists of larger number of registers.
- It consists of less number of transistors.

CISC Processor

CISC stands for **Complex Instruction Set Computer**. It is designed to minimize the number of instructions per program, ignoring the number of cycles per instruction. The emphasis is on building complex instructions directly into the hardware.

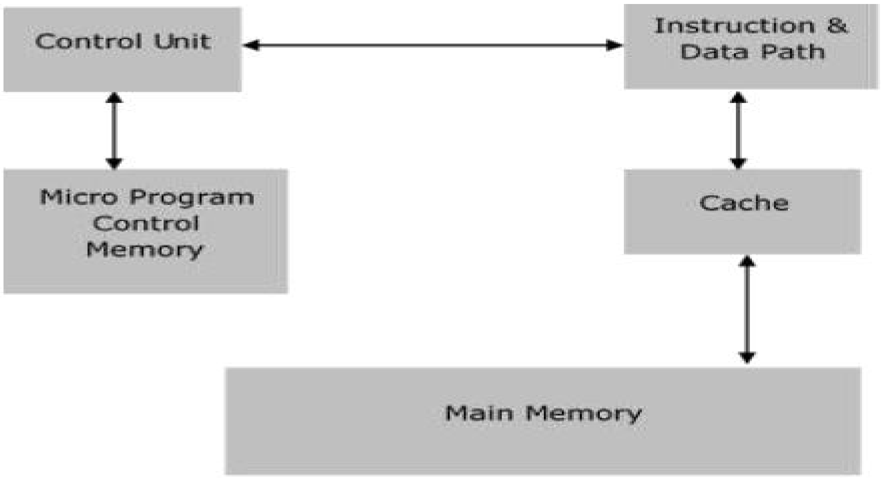
The compiler has to do very little work to translate a high-level language into assembly level language/machine code because the length of the code is relatively short, so very little RAM is required to store the instructions.

Some of the CISC Processors are −

- IBM 370/168
- VAX 11/780
- Intel 80486

Architecture of CISC

Its architecture is designed to decrease the memory cost because more storage is needed in larger programs resulting in higher memory cost. To resolve this, the number of instructions per program can be reduced by embedding the number of operations in a single instruction.



Characteristics of CISC

- Variety of addressing modes.
- Larger number of instructions.
- Variable length of instruction formats.
- Several cycles may be required to execute one instruction.
- Instruction-decoding logic is complex.
- One instruction is required to support multiple addressing modes.

Special Processors

These are the processors which are designed for some special purposes. Few of the special processors are briefly discussed −

Coprocessor

A coprocessor is a specially designed microprocessor, which can handle its particular function many times faster than the ordinary microprocessor.

For example − Math Coprocessor.

Some Intel math-coprocessors are −

- 8087-used with 8086
- 80287-used with 80286
- 80387-used with 80386

Input/Output Processor

It is a specially designed microprocessor having a local memory of its own, which is used to control I/O devices with minimum CPU involvement.

For example −

- DMA (direct Memory Access) controller
- Keyboard/mouse controller
- Graphic display controller
- SCSI port controller

Transputer (Transistor Computer)

A transputer is a specially designed microprocessor with its own local memory and having links to connect one transputer to another transputer for inter-processor communications. It was first designed in 1980 by Inmos and is targeted to the utilization of VLSI technology.

A transputer can be used as a single processor system or can be connected to external links, which reduces the construction cost and increases the performance.

For example − 16-bit T212, 32-bit T425, the floating point (T800, T805 & T9000) processors.

DSP (Digital Signal Processor)

This processor is specially designed to process the analog signals into a digital form. This is done by sampling the voltage level at regular time intervals and converting the voltage at that instant into a digital form. This process is performed by a circuit called an analogue to digital converter, A to D converter or ADC.

A DSP contains the following components −

- **Program Memory** − It stores the programs that DSP will use to process data.
- **Data Memory** − It stores the information to be processed.
- **Compute Engine** − It performs the mathematical processing, accessing the program from the program memory and the data from the data memory.
- **Input/Output** − It connects to the outside world.

Its applications are −

- Sound and music synthesis
- Audio and video compression
- Video signal processing
- 2D and 3d graphics acceleration.

For example − Texas Instrument’s TMS 320 series, e.g., TMS 320C40, TMS320C50.

Microprocessor - 8085 Architecture

085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8-bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration −

- 8-bit data bus
- 16-bit address bus, which can address upto 64KB
- A 16-bit program counter
- A 16-bit stack pointer
- Six 8-bit registers arranged in pairs: BC, DE, HL
- Requires +5V supply to operate at 3.2 MHZ single phase clock

It is used in washing machines, microwave ovens, mobile phones, etc.

8085 Microprocessor – Functional Units

8085 consists of the following functional units −

Accumulator

It is an 8-bit register used to perform arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU.

Arithmetic and logic unit

As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

General purpose register

There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H & L. Each register can hold 8-bit data.

These registers can work in pair to hold 16-bit data and their pairing combination is like B-C, D-E & H-L.

Program counter

It is a 16-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

Stack pointer

It is also a 16-bit register works like stack, which is always incremented/decremented by 2 during push & pop operations.

Temporary register

It is an 8-bit register, which holds the temporary data of arithmetic and logical operations.

Flag register

It is an 8-bit register having five 1-bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.

These are the set of 5 flip-flops −

- Sign (S)
- Zero (Z)
- Auxiliary Carry (AC)
- Parity (P)
- Carry (C)

Its bit position is shown in the following table −

D7	D6	D5	D4	D3	D2	D1	D0
S	Z		AC		P		CY

Instruction register and decoder

It is an 8-bit register. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder decodes the information present in the Instruction register.

Timing and control unit

It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals, which control external and internal circuits −

- Control Signals: READY, RD’, WR’, ALE
- Status Signals: S0, S1, IO/M’
- DMA Signals: HOLD, HLDA
- RESET Signals: RESET IN, RESET OUT

Interrupt control

As the name suggests it controls the interrupts during a process. When a microprocessor is executing a main program and whenever an interrupt occurs, the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control goes back to the main program.

There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

Serial Input/output control

It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).

Address buffer and address-data buffer

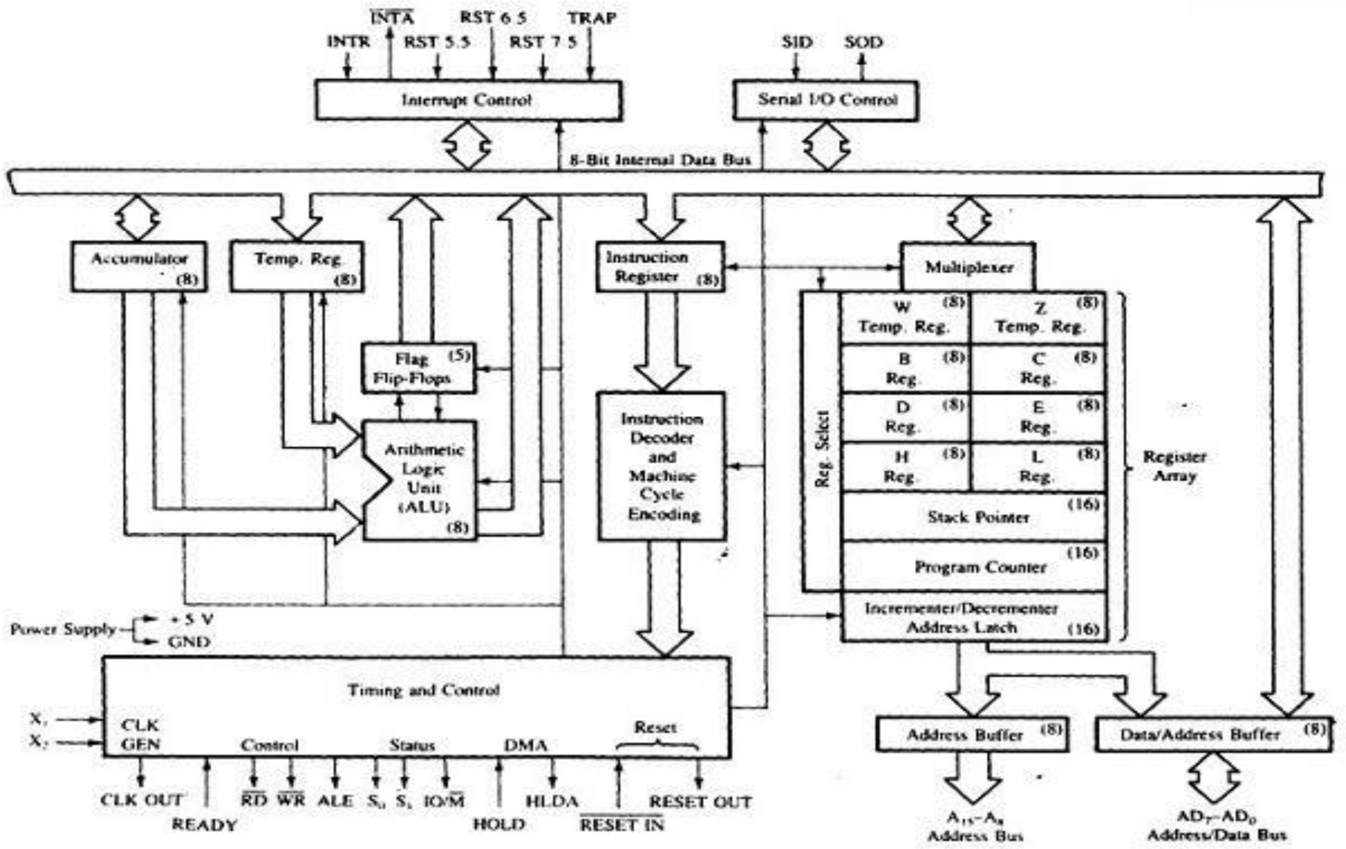
The content stored in the stack pointer and program counter is loaded into the address buffer and address-data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

Address bus and data bus

Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

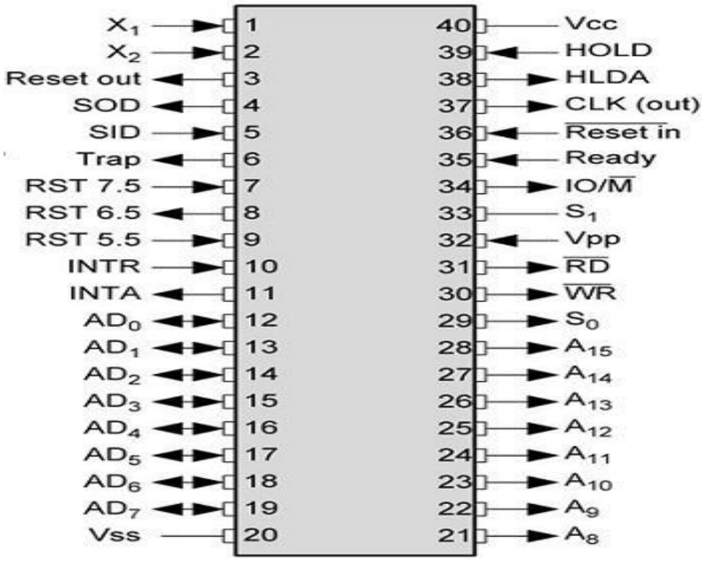
8085 Architecture

We have tried to depict the architecture of 8085 with this following image −



Microprocessor - 8085 Pin Configuration

The following image depicts the pin diagram of 8085 Microprocessor −



The pins of a 8085 microprocessor can be classified into seven groups −

Address bus

A15-A8, it carries the most significant 8-bits of memory/IO address.

Data bus

AD7-AD0, it carries the least significant 8-bit address and data bus.

Control and status signals

These signals are used to identify the nature of operation. There are 3 control signal and 3 status signals.

Three control signals are RD, WR & ALE.

- **RD** − This signal indicates that the selected IO or memory device is to be read and is ready for accepting data available on the data bus.
- **WR** − This signal indicates that the data on the data bus is to be written into a selected memory or IO location.
- **ALE** − It is a positive going pulse generated when a new operation is started by the microprocessor. When the pulse goes high, it indicates address. When the pulse goes down it indicates data.

Three status signals are IO/M, S0 & S1.

IO/M

This signal is used to differentiate between IO and Memory operations, i.e. when it is high indicates IO operation and when it is low then it indicates memory operation.

S1 & S0

These signals are used to identify the type of current operation.

Power supply

There are 2 power supply signals − VCC & VSS. VCC indicates +5v power supply and VSS indicates ground signal.

Clock signals

There are 3 clock signals, i.e. X1, X2, CLK OUT.

- **X1, X2** − A crystal (RC, LC N/W) is connected at these two pins and is used to set frequency of the internal clock generator. This frequency is internally divided by 2.
- **CLK OUT** − This signal is used as the system clock for devices connected with the microprocessor.

Interrupts & externally initiated signals

Interrupts are the signals generated by external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. We will discuss interrupts in detail in interrupts section.

- **INTA** − It is an interrupt acknowledgment signal.
- **RESET IN** − This signal is used to reset the microprocessor by setting the program counter to zero.
- **RESET OUT** − This signal is used to reset all the connected devices when the microprocessor is reset.
- **READY** − This signal indicates that the device is ready to send or receive data. If READY is low, then the CPU has to wait for READY to go high.
- **HOLD** − This signal indicates that another master is requesting the use of the address and data buses.
- **HLDA (HOLD Acknowledge)** − It indicates that the CPU has received the HOLD request and it will relinquish the bus in the next clock cycle. HLDA is set to low after the HOLD signal is removed.

Serial I/O signals

There are 2 serial signals, i.e. SID and SOD and these signals are used for serial communication.

- **SOD** (Serial output data line) − The output SOD is set/reset as specified by the SIM instruction.

- **SID** (Serial input data line) − The data on this line is loaded into accumulator whenever a RIM instruction is executed.

Microprocessor - 8086 Overview

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

The most prominent features of a 8086 microprocessor are as follows −

- It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- It is available in 3 versions based on the frequency of operation −
    - 8086 → 5MHz
    - 8086-2 → 8MHz
    - (c)8086-1 → 10 MHz
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

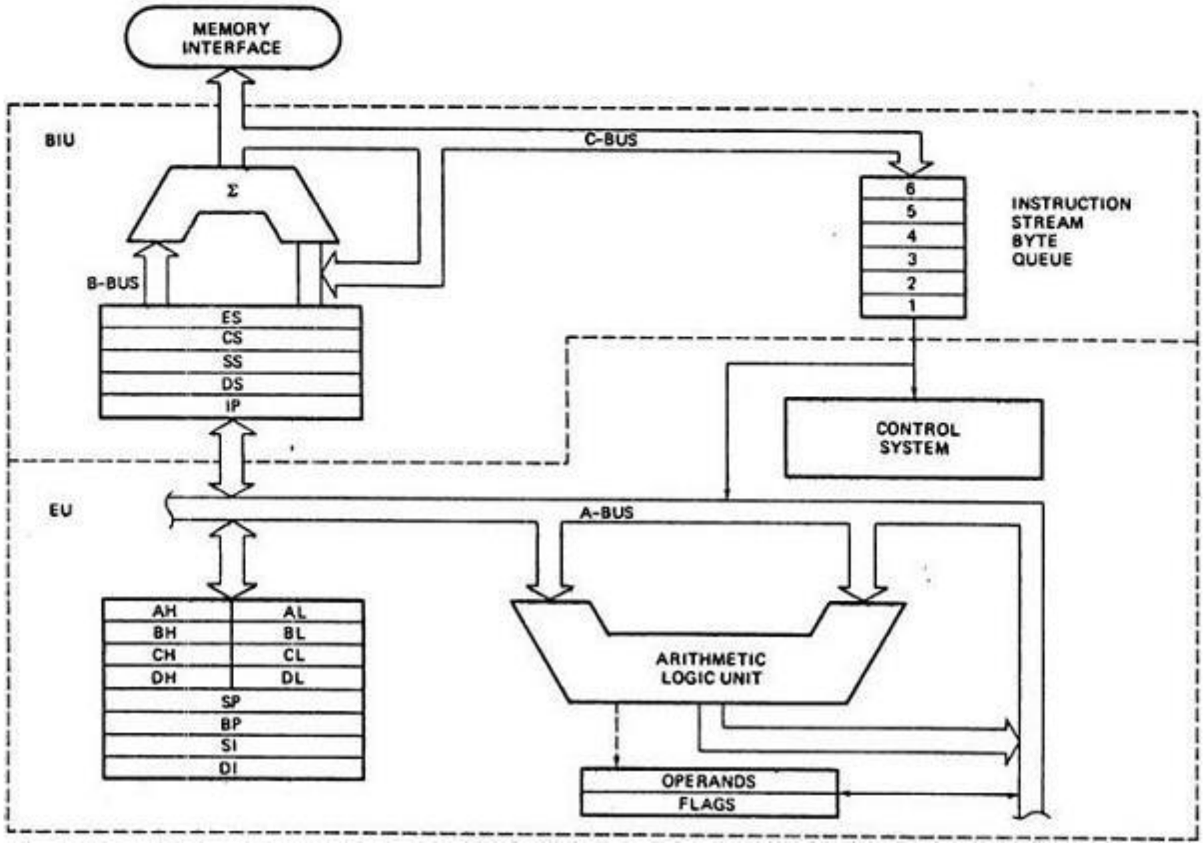
Comparison between 8085 & 8086 Microprocessor

- **Size** − 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- **Address Bus** − 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- **Memory** − 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- **Instruction** − 8085 doesn’t have an instruction queue, whereas 8086 has an instruction queue.

- **Pipelining** − 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.
- **I/O** − 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- **Cost** − The cost of 8085 is low whereas that of 8086 is high.

Architecture of 8086

The following diagram depicts the architecture of a 8086 Microprocessor −



Microprocessor - 8086 Functional Units

8086 Microprocessor is divided into two functional units, i.e., **EU** (Execution Unit) and **BIU** (Bus Interface Unit).

EU (Execution Unit)

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

Let us now discuss the functional parts of 8086 microprocessors.

ALU

It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

Flag Register

It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.

Conditional Flags

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags −

- **Carry flag** − This flag indicates an overflow condition for arithmetic operations.
- **Auxiliary flag** − When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- **Parity flag** − This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.
- **Zero flag** − This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- **Sign flag** − This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.
- **Overflow flag** − This flag represents the result when the system capacity is exceeded.

Control Flags

Control flags controls the operations of the execution unit. Following is the list of control flags −

- **Trap flag** − It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.
- **Interrupt flag** − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

- **Direction flag** − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General purpose register

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

- **AX register** − It is also known as accumulator register. It is used to store operands for arithmetic operations.
- **BX register** − It is used as a base register. It is used to store the starting base address of the memory area within the data segment.
- **CX register** − It is referred to as counter. It is used in loop instruction to store the loop counter.
- **DX register** − This register is used to hold I/O port address for I/O instruction.

Stack pointer register

It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.

BIU (Bus Interface Unit)

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

It has the following functional parts −

- **Instruction queue** − BIU contains the instruction queue. BIU gets upto 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
- Fetching the next instruction while the current instruction executes is called **pipelining**.
- **Segment register** − BIU has 4 segment buses, i.e. CS, DS, SS& ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to executed by the EU.
    - **CS** − It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

  - **DS** − It stands for Data Segment. It consists of data used by the program andis accessed in the data segment by an offset address or the content of other register that holds the offset address.
  - **SS** − It stands for Stack Segment. It handles memory to store data and addresses during execution.
  - **ES** − It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.
- **Instruction pointer** − It is a 16-bit register used to hold the address of the next instruction to be executed.

Register organization of 8086

General 16-bit registers
The registers AX, BX, CX, and DX are the general 16-bit registers.

AX Register: Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16- bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations, rotate and string manipulation.

BX Register: This register is mainly used as a base register. It holds the starting base location of a memory region within a data segment. It is used as offset storage for forming physical address in case of certain addressing mode.

CX Register: It is used as default counter or count register in case of string and loop instructions.

DX Register: Data register can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

Segment registers:
To complete 1Mbyte memory is divided into 16 logical segments. The complete 1Mbyte memory segmentation is as shown in fig 1.5. Each segment contains 64Kbyte of memory. There are four segment registers.

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

Stack segment (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction. It is used for addressing stack segment of memory. The stack segment is that segment of memory, which is used to store stack data.

Data segment (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions. It points to the data segment memory where the data is resided.

Extra segment (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It also refers to segment which essentially is another data segment of the memory. It also contains data.

Pointers and index registers.

The pointers contain within the particular segments. The pointers IP, BP, SP usually contain offsets within the code, data and stack segments respectively
Stack Pointer (SP) is a 16-bit register pointing to program stack in stack segment.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Conditional Flags

Conditional flags are as follows:
Carry Flag (CY): This flag indicates an overflow condition for unsigned integer arithmetic. It is also used in multiple-precision arithmetic.

Auxiliary Flag (AC): If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), the AC flag is set i.e. carry given by D3 bit to D4 is AC flag. This is not a general-purpose flag, it is used internally by the Processor to perform Binary to BCD conversion.

Parity Flag (PF): This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1's, the Parity Flag is set and for odd number of 1's, the Parity flag is reset.

Zero Flag (ZF): It is set; if the result of arithmetic or logical operation is zero else it is reset.

Sign Flag (SF): In sign magnitude format the sign of number is indicated by MSB bit. If the result of operation is negative, sign flag is set.

Control Flags

Control flags are set or reset deliberately to control the operations of the execution unit.

Control flags are as follows:

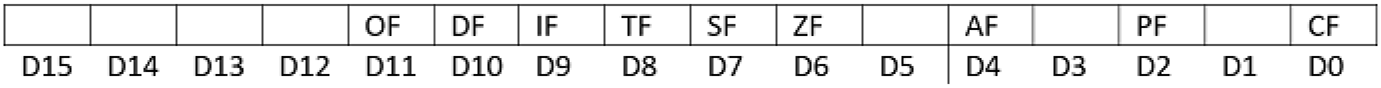
Trap Flag (TF): It is used for single step control. It allows user to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

Interrupt Flag (IF): It is an interrupt enable/disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction sit and can be cleared by executing CLI instruction.

Direction Flag (DF): It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

Flag Register in 8086 Microprocessor

Flag Register is a 16-bit register, but there are only 9 flags available in the 8086 microprocessor. The rest 7 bits are hence left idle.



				OF	DF	IF	TF	SF	ZF		AF		PF		CF
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

There are two **categories of flag register**:

1. Condition flags
2. Control flags

1) Condition flags

The conditional flags are set or reset after any arithmetic or logical operation is performed on an 8 bit or 16-bit number. This category consists of the following 6 flags:

i. **Carry Flag (CF):** The carry flag will be set only if a carry is generated from the MSB of the result after doing any operation in 8086 Microprocessor.
ii. **Parity Flag (PF):** Parity is related to the number of 1's contained in the binary data. There exist two types of parity:
    - **Even Parity:** When the number of 1's in the binary data are even.
    - **Odd Parity:** When the number of 1's in the binary data are odd.

    For the flag, the PF is set if there exists an even parity in data after the execution of the instruction. Else the flag is reset.

iii. **Auxiliary-Carry Flag (AF):** This flag is set if there is a generation of carrying from a nibble, i.e. 4 bits of data.
iv. **Zero Flag (ZF):** If the result after performing the required operation (Arithmetic or Logical) on the instructions is zero, in that case, the zero flags are set to 1. Else, it remains reset.
v. **Sign Flag (SF):** If the result after performing any arithmetic or logic operation in the given instruction is negative, then the sign flag is set to 1. Else, for a positive result, the sign flag remains reset.
vi. **Overflow Flag (OF):** This Flag will be set if the register gets overflowed with data after any arithmetic or logic operation. This happens in cases when the carry is getting in in MSB, but there is no space in the register to store the carried out bit.

2) Control flags

The control flags are used to navigate the microprocessor for certain operations. There are 3 types of control flags:

i. **Trap Flag (TF):** This flag is used of we need single-step debugging in our code. If the TF is set, then the execution will be done step by step. Otherwise, the free-running operation will be done.

ii. **Interrupt Flag (IF):** This flag is used to enable the Interrupt. The microprocessor is capable of handling interrupts only if this flag is in the set mode. Otherwise, any interrupt raised while the execution of the instructions will not be handled by the microprocessor.

iii. **Direction Flag (DF):** This flag is used for string operations. If this flag is set, the string will be read from higher-order bits to lower order bits and vice versa.

Microprocessor - 8086 Addressing Modes

The different ways in which a source operand is denoted in an instruction is known as **addressing modes**. There are 8 different addressing modes in 8086 programming −

Immediate addressing mode

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example

MOV CX, 4929 H,
ADD AX, 2387 H,
MOV AL, FFH

Register addressing mode

It means that the register is the source of an operand for an instruction.

Example

```
MOV CX, AX   ; copies the contents of the 16-bit AX register into
         ; the 16-bit CX register),
ADD BX, AX
```

Direct addressing mode

The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example

MOV AX, [1592H],
MOV AL, [0300H]

Register indirect addressing mode

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example

```
MOV AX, [BX]  ; Suppose the register BX contains 4895H, then the contents
              ; 4895H are moved to AX
ADD CX, {BX}
```

Based addressing mode

In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

Example

```
MOV DX, [BX+04], ADD CL, [BX+08]
```

Indexed addressing mode

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

Example

```
MOV BX, [SI+16],
 ADD AL, [DI+16]
```

Based-index addressing mode

In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

Example

```
ADD CX, [AX+SI],
MOV AX, [AX+DI]
```

Based indexed with displacement mode

In this addressing mode, the operands offset is computed by adding the base register contents. An Index registers contents and 8 or 16-bit displacement.

Example

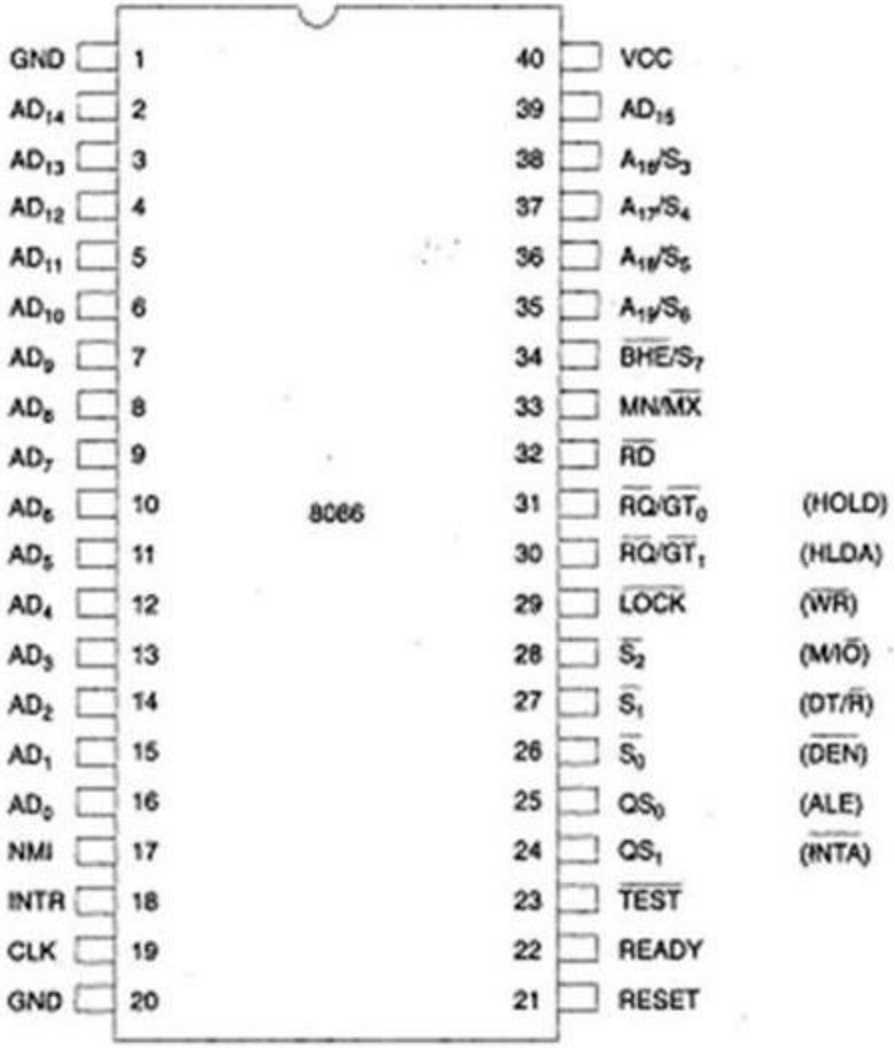
```
MOV AX, [BX+DI+08],
ADD CX, [BX+SI+16]
```

Microprocessor - 8086 Pin Configuration

8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip. Let us now discuss in detail the pin configuration of a 8086 Microprocessor.

8086 Pin Diagram

Here is the pin diagram of 8086 microprocessor −



Let us now discuss the signals in detail −

Power supply and frequency signals

It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

Clock signal

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus

AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus

A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

S7/BHE

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

Read(*$\overline{RD}$*)

It is available at pin 32 and is used to read signal for Read operation.

Ready

It is available at pin 22. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

RESET

It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

$\overline{TEST}$

This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

MN/*$\overline{MX}$*

It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-aversa.

INTA

It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.

DT/R

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-a-versa.

M/IO

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR

It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

HLDA

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

QS$_1$ and QS$_0$

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table −

QS$_0$	QS$_1$	Status
0	0	No operation
0	1	First byte of opcode from the queue

1	0	Empty the queue
1	1	Subsequent byte from the queue

S_0, S_1, S_2

These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status −

S_2	**S_1**	**S_0**	**Status**
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

LOCK

When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

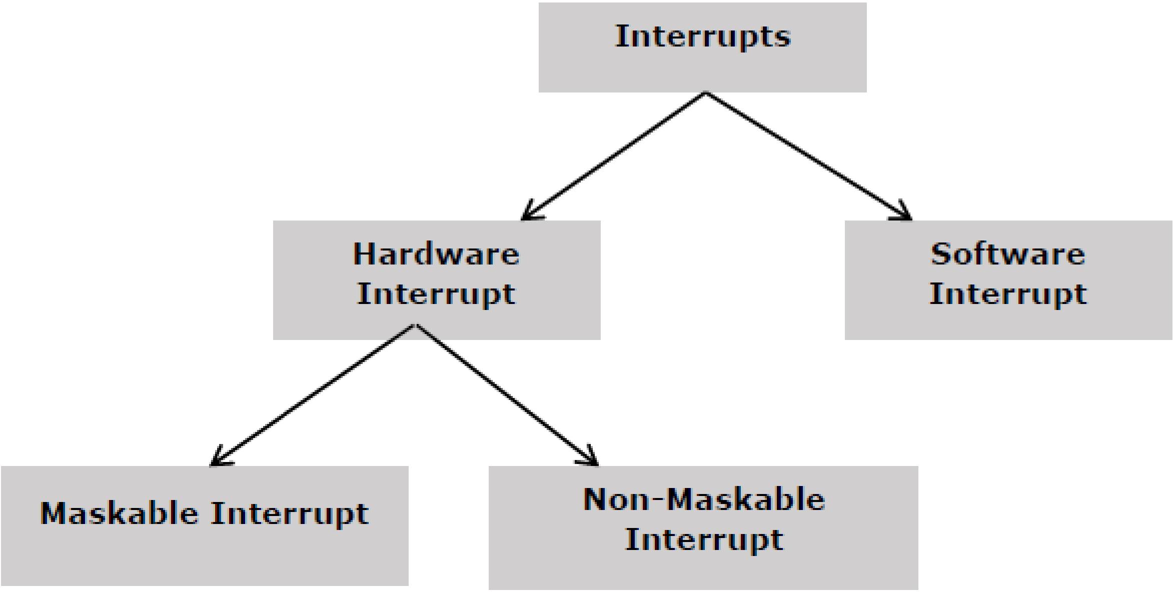
RQ/GT$_1$ and RQ/GT$_0$

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT$_0$ has a higher priority than RQ/GT$_1$.

Microprocessor - 8086 Interrupts

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an **ISR** (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

The following image shows the types of interrupts we have in a 8086 microprocessor −



Hardware Interrupts

Hardware interrupt is caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

The 8086 has two hardware interrupt pins, i.e. NMI and INTR. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority. One more interrupt pin associated is INTA called interrupt acknowledge.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR)and it is of type 2 interrupt.

When this interrupt is activated, these actions take place −

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.

- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.
- Interrupt flag and trap flag are reset to 0.

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice. The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor −

- First completes the current instruction.
- Activates INTA output and receives the interrupt type, say X.
- Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.
- IP value is loaded from the contents of word location X × 4
- CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INT- Interrupt instruction with type number

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number' × 4
- CS is loaded from the contents of the next word location.

- Interrupt Flag and Trap Flag are reset to 0

The starting address for type0 interrupt is 000000H, for type1 interrupt is 00004H similarly for type2 is 00008H and ……so on. The first five pointers are dedicated interrupt pointers. i.e. −

- **TYPE 0** interrupt represents division by zero situation.
- **TYPE 1** interrupt represents single-step execution during the debugging of a program.
- **TYPE 2** interrupt represents non-maskable NMI interrupt.
- **TYPE 3** interrupt represents break-point interrupt.
- **TYPE 4** interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

INT 3-Break Point Interrupt Instruction

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps −

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 3×4 = 0000CH
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction

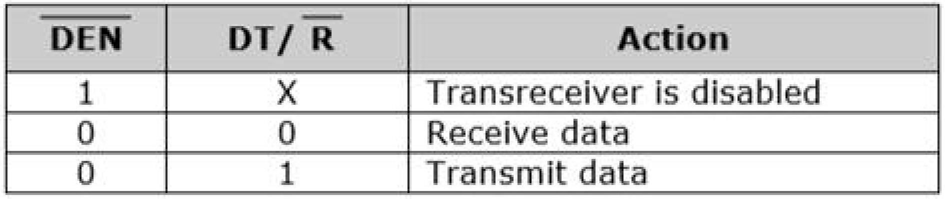
It is a 1-byte instruction and their mnemonic **INTO**. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

Its execution includes the following steps −

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location 4×4 = 00010H
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

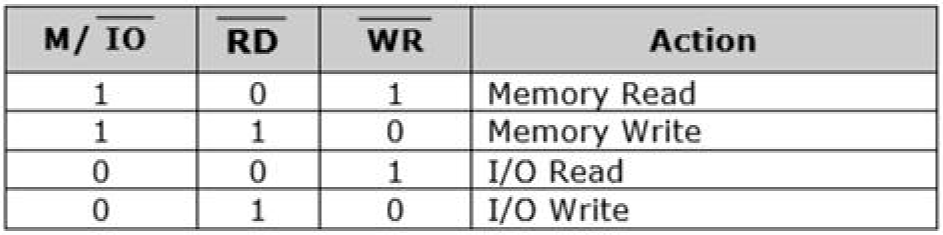
MINIMUM MODE OPERATIONS OF 8086:

- 8086 works in Minimum Mode, when MN/ ¯MX = 1.
- Minimum Mode, 8086 is the only processor in the system. The Minimum Mode circuit of 8086 is as shown below:
- Clock is provided by the 8284 clock generator, it provides CLK, RESET and READY input to 8086.
- Address from the address bus is latched into 8282 8-bit latch. Three such latches are needed, as address bus is 20-bit. The ALE of 8086 is connected to STB of the latch. The ALE for this latch is given by 8086 itself.
- The data bus is driven through 8286 8-bit trans-receiver. Two such trans-receivers are needed, as the data bus is 16-bit. The trans-receivers are enabled through the DEN signal, while the direction of data is controlled by the DT/ ¯R signal. ¯DEN is connected to ¯OE and DT/ ¯R is connected to T. Both ¯DEN and DT/ ¯R are given by 8086 itself.



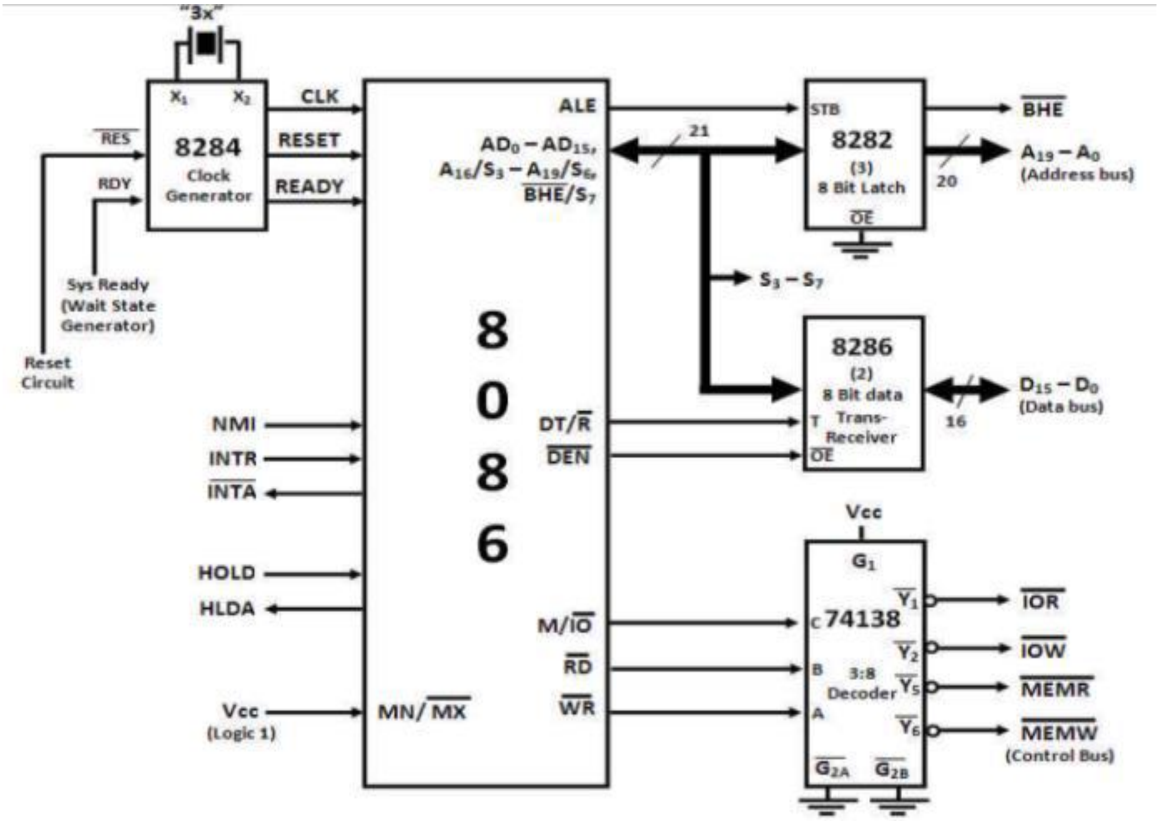
$\overline{\text{DEN}}$	DT/ $\overline{\text{R}}$	Action
1	X	Transreceiver is disabled
0	0	Receive data
0	1	Transmit data

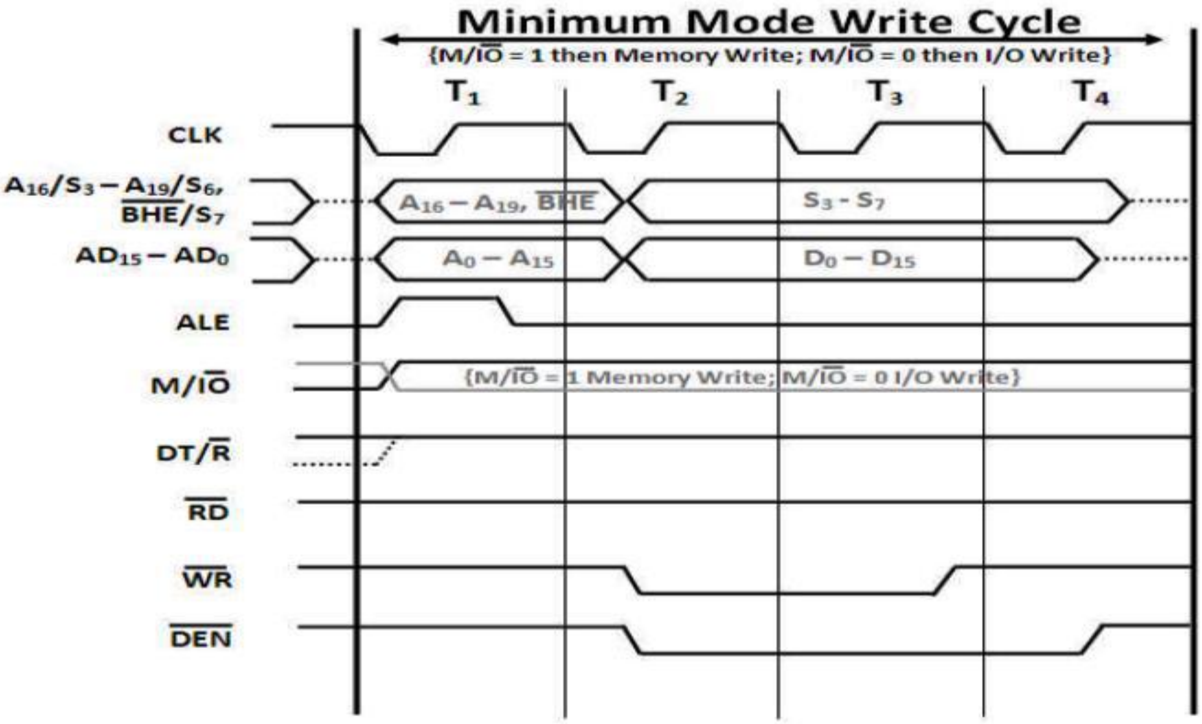
- Control signals for all operations are generated by decoding M/¯IO , ¯RD and ¯WR signals.



M/ $\overline{\text{IO}}$	$\overline{\text{RD}}$	$\overline{\text{WR}}$	Action
1	0	1	Memory Read
1	1	0	Memory Write
0	0	1	I/O Read
0	1	0	I/O Write

- M/¯IO , ¯RD and ¯WR are decoded by a 3:8 decoder like IC 74138. Bus Request (DMA) is done using the HOLD and HLDA signals.
- ¯INTA is given by 8086, in response to an interrupt on INTR line.





MAXIMUM MODE OPERATION OF 8086:

8086 microprocessor characteristics:

- It contains 20 bit address bus.
- It contains 16-bit data bus, therefore 8086 is called as **16-bit microprocessor.**
- It is 2-stage pipelined processor. It can prefetch 6 bytes from memory and store into queue to increase the speed of the execution.
- It's control bus carries signals for executing operations such as read ,write etc.
- It has Memory Banks. 2 banks of 512KB each. These banks are called as lower Bank (even) and higher Bank (odd).
- In 8086 the entire memory is divided into four memory segments which are code ,stack, data and extra segment.
- 8086 has 16 bit IO address.
- It has 256 interrupts.

8086 has two operating Modes:

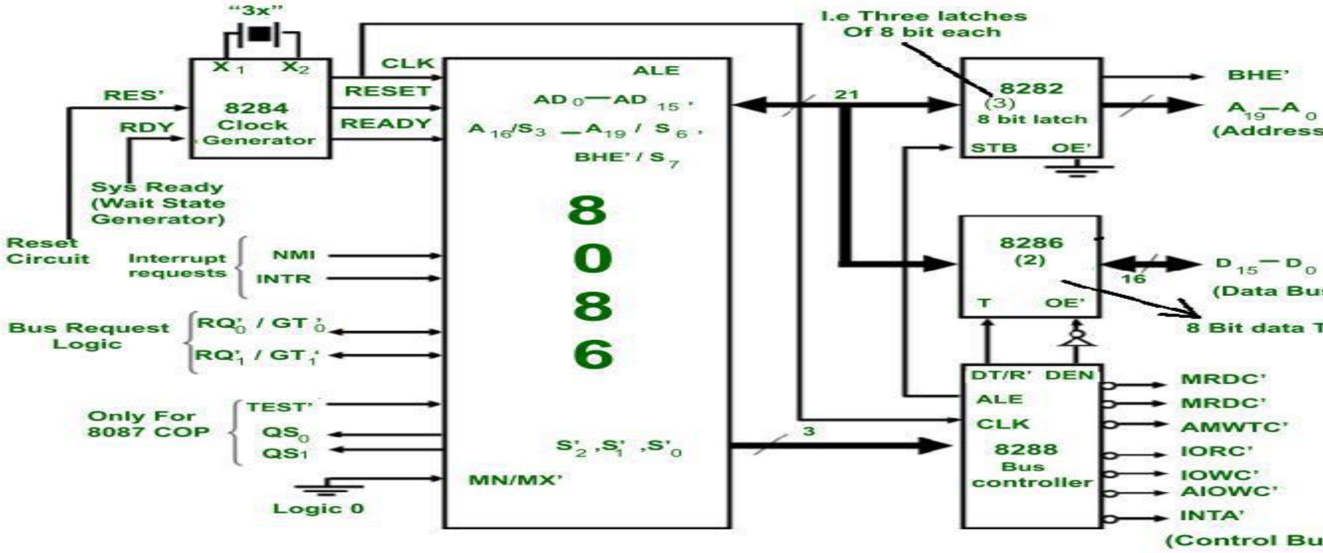
1. Minimum mode
2. Maximum mode

Minimum mode:

- In this 8086 is the only processor in the system . In a minimum mode 8086 system.
- 8086 is operated in minimum mode when MN/MX' pin to logic 1.
- In this mode, all the control signals are given out by the 8086 itself.

Maximum mode:

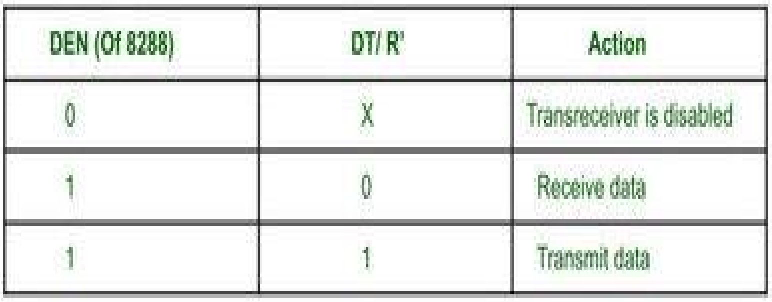
- In this we can connect more processors to 8086 (8087/8089).
- 8086 max mode is basically for implementation of allocation of global resources and passing bus control to other coprocessor(i.e. second processor in the system), because two processors can not access system bus at same instant.
- All processors execute their own program.
- The resources which are common to all processors are known as global resources.
- The resources which are allocated to a particular processor are known as local or private resources.



Maximum mode circuit

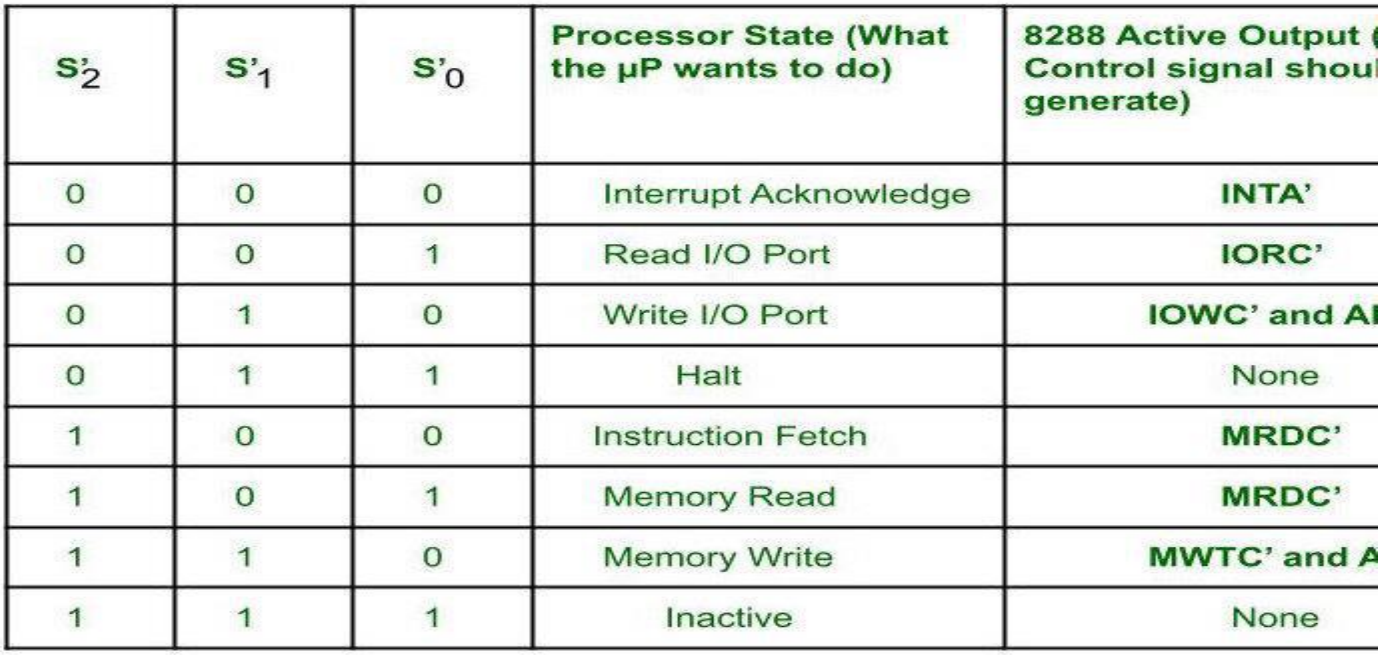
Circuit explanation:

- When MN/ MX' = 0 , 8086 works in max mode.
- Clock is provided by **8284 clock generator.**
- **8288 bus controller-** Address form the address bus is latched into 8282 8-bit latch. Three such latches are required because **address bus is 20 bit**. The ALE(Address latch enable) is connected to STB(Strobe) of the latch. **The ALE for latch is given by 8288 bus controller.**
- The data bus is operated through 8286 8-bit transceiver. Two such transceivers are required, because **data bus is 16-bit**. The transceivers are enabled the DEN signal, while the direction of data is controlled by the DT/R signal. DEN is connected to **OE'** and **DT/ R'** is connected to T. **Both DEN and DT/ R' are given by 8288 bus controller.**



DEN (Of 8288)	DT/ R'	Action
0	X	Transreceiver is disabled
1	0	Receive data
1	1	Transmit data

- **Control signals for all operations are generated by decoding S'$_2$, S'$_1$ and S'$_0$ using 8288 bus controller.**



S'$_2$	S'$_1$	S'$_0$	Processor State (What the µP wants to do)	8288 Active Output (Control signal shoul generate)
0	0	0	Interrupt Acknowledge	INTA'
0	0	1	Read I/O Port	IORC'
0	1	0	Write I/O Port	IOWC' and AI
0	1	1	Halt	None
1	0	0	Instruction Fetch	MRDC'
1	0	1	Memory Read	MRDC'
1	1	0	Memory Write	MWTC' and A
1	1	1	Inactive	None

- Bus request is done using RQ' / GT' lines interfaced with 8086. RQ$_0$/GT$_0$ has more priority than RQ$_1$/GT$_1$.
- INTA' is given by 8288, in response to an interrupt on INTR line of 8086.
- In max mode, the advanced write signals get enabled one T-state in advance as compared to normal write signals. This gives slower devices more time to get ready to accept the data, therefore it reduces the number of cycles.

APPLICATIONS OF MICROPROCESSOR:

- The microprocessor is used in personal computers (PCs).
- The microprocessor is used in LASER printers for good speed and making automatic photo copies.
- The microprocessors are used in modems, telephone, digital telephone sets, and also in air reservation systems and railway reservation systems.
- The microprocessor is used in medical instrument to measure temperature and blood pressure.
- It is also used in mobile phones and television.
- It is used in calculators and game machine.
- It is used in accounting system and data acquisition system.
- It is used in military applications.
- It is also used in traffic light control.

UNIT V

MICROPROCESSORS-II

ADDRESSING MODES OF 8086:

Addressing modes indicates way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction.

According to the flow of instruction execution, the instruction may be categorized as:

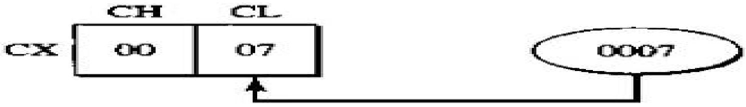
Sequential Control flow instructions   Control Transfer instructions

- **Sequential Control flow instructions:** In this type of instruction after execution control can be transferred to the next immediately appearing instruction in the program.

  The addressing modes for sequential control transfer instructions are as follows:

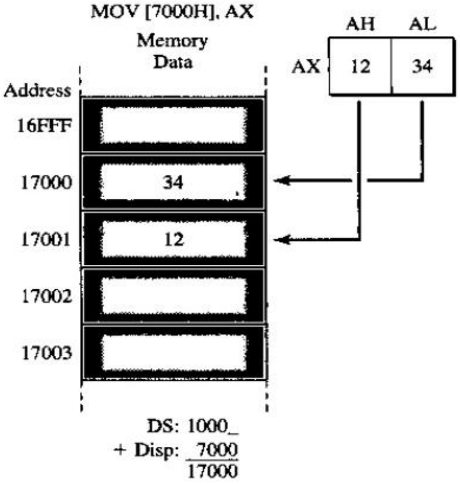
- **Immediate addressing mode:** In this mode, immediate is a part of instruction and appears in the form of successive byte or bytes.

  Example: MOV CX, 0007H; Here 0007 is the immediate data



- **Direct Addressing mode:** In this mode, the instruction operand specifies the memory address where data is located.

  Example: MOV AX, [5000H]; Data is available in 5000H memory location



Effective Address (EA) is computed using 5000H as offset address and content of DS as segment address.

EA=10H * DS + 5000H

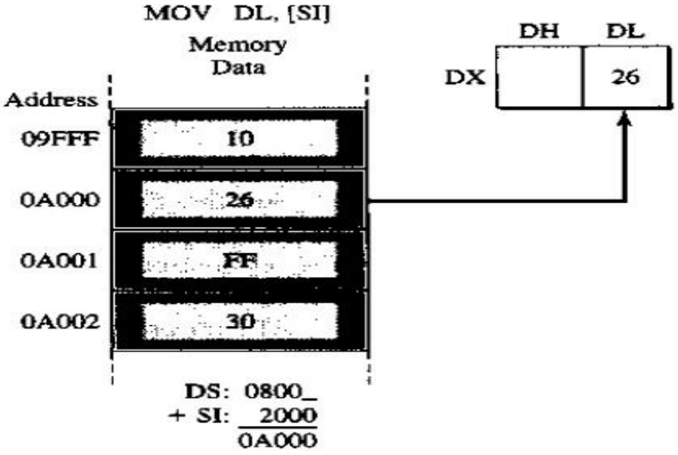
- **Register Addressing mode:** In this mode, the data is stored in a register and it is referred using particular register. All the registers except IP may be used in this mode.

  Example: MOV AX, BX;

- **Register Indirect addressing mode:** In this mode, instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.

  Example: MOV AX, [BX];   **EA=10H * DS + [BX]**

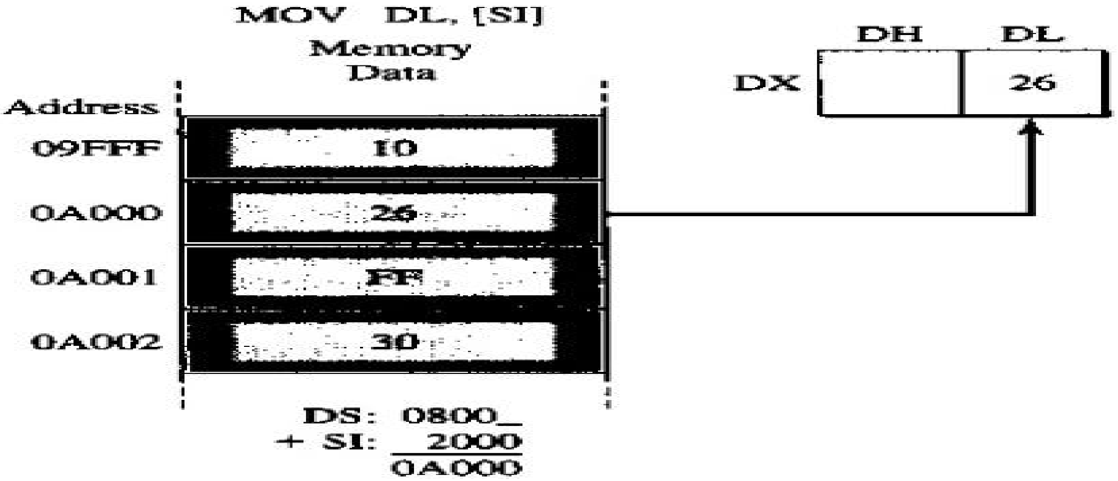
- **Indexed Addressing mode:** 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides. DS and ES are default segments for index registers SI and DI. DS=0800H, SI=2000H, MOV DL, [SI]



Example: MOV AX, [SI]; **EA=10H * DS + [SI]**

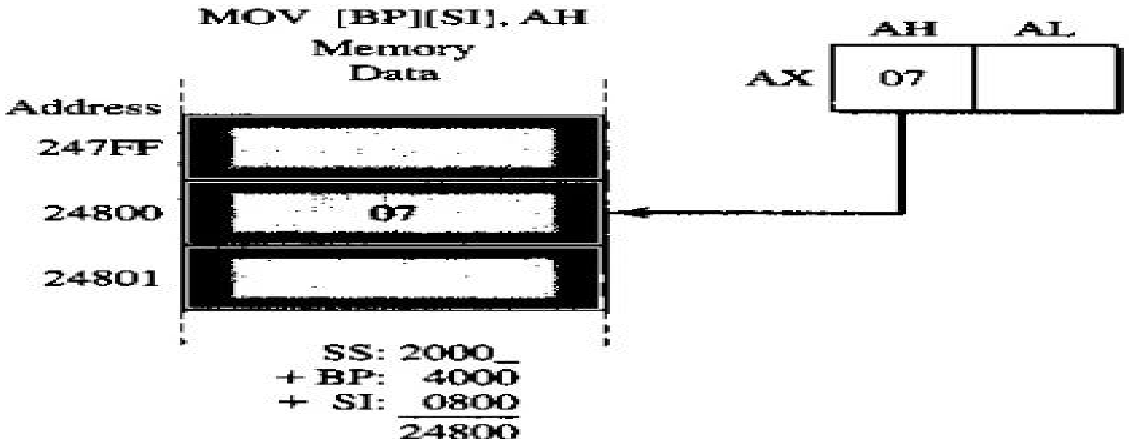
- **Register Relative Addressing mode:** In this mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI, DI in the default segments.

Example: MOV AX, 50H [BX]; **EA=10H * DS + 50H + [BX]**

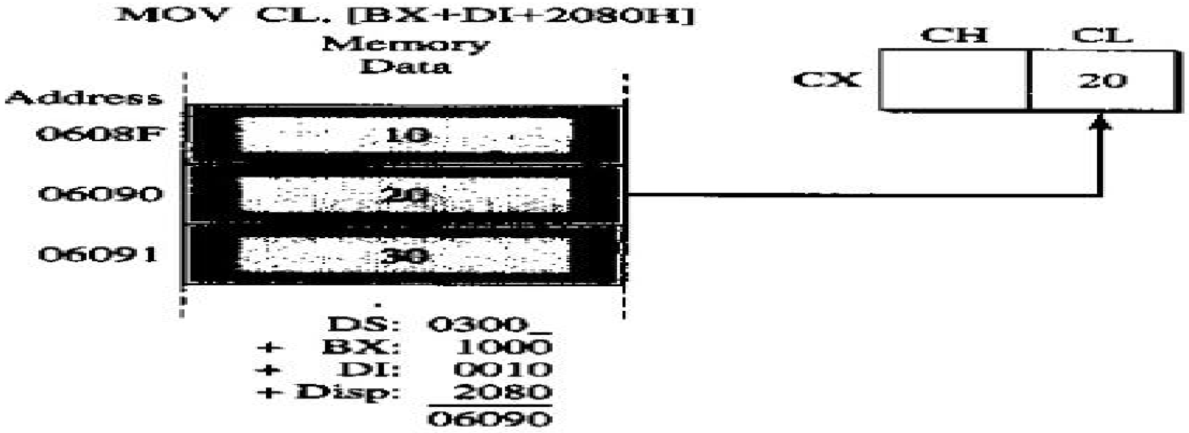


- **Based Indexed Addressing mode:** In this mode, the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

Example: MOV AX, [BX] [SI]; **EA=10H * DS + [BX] + [SI]**



- **Relative Based Indexed Addressing mode:** In this mode, 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.
  Example: MOV AX, 50H [BX] [SI]; **EA=10H * DS + 50H + [BX] + [SI]**



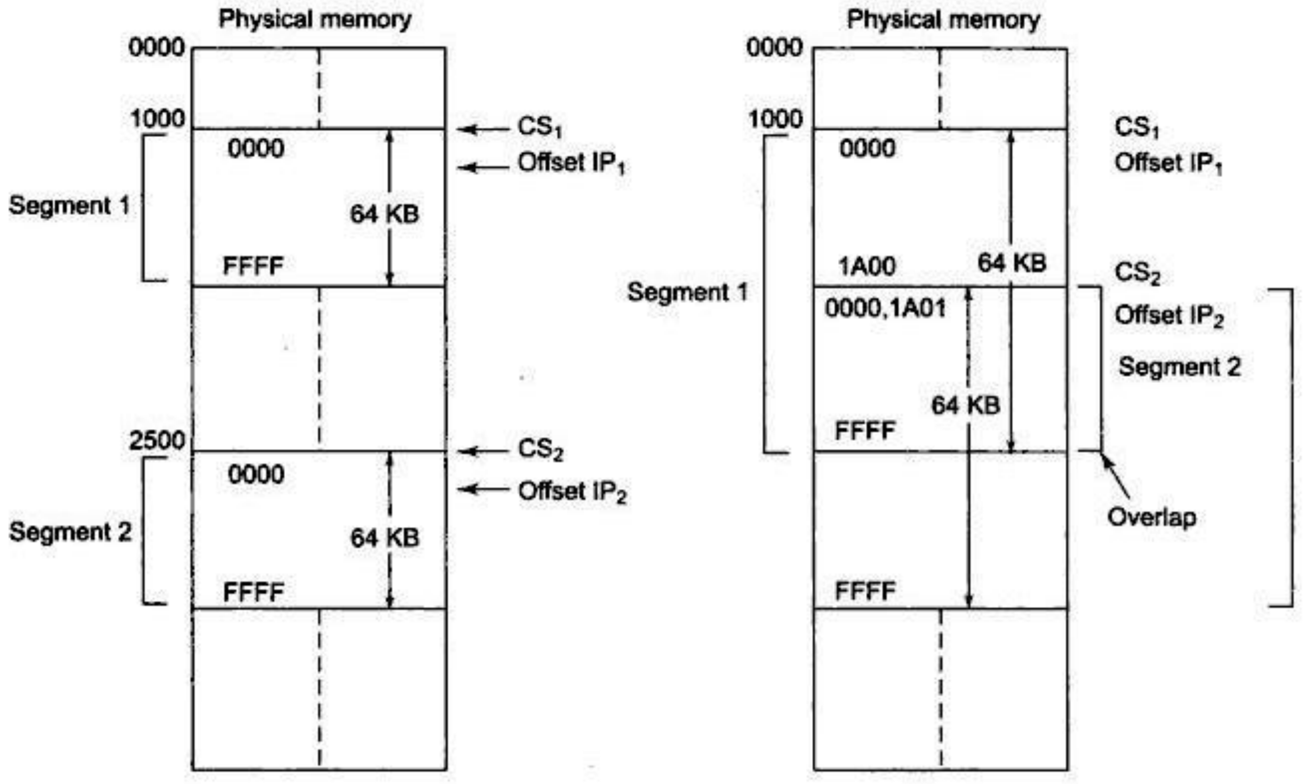
- **Control Transfer Instructions:** In control transfer instruction, the control can be transferred to some predefined address or the address somehow specified in the instruction after their execution.
  For the control transfer instructions, the addressing modes depend upon whether the destination location is within the segment or different segments. It also depends upon the method of passing the destination address to the processor. Depending on this control transfer instructions are categorized as follows:
- **Intra segment Direct mode:** In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.
- **Intra segment Indirect mode:** In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies but it is passed to the instruction indirectly.
- **Inter segment Direct mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.
- **Inter segment Indirect mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies but it is passed to the instruction indirectly.

Memory Segmentation for 8086:

8086, via its 20-bit address bus, can address 220 = 1,048,576 or 1 MB of different memory locations. Thus the memory space of 8086 can be thought of as consisting of 1,048,576 bytes or 524,288 words. The memory map of 8086 is shown in Figure where the whole memory space starting from 00000 H to FFFFF H is divided into 16 blocks—each one consisting of 64KB.

1 MB memory of 8086 is partitioned into 16 segments—each segment is of 64 KB length. Out of these 16 segments, only 4 segments can be active at any given instant of time— these are code segment, stack segment, data segment and extra segment. The four memory segments that the CPU works with at any time are called currently active segments. Corresponding to these four segments, the registers used are Code Segment Register (CS), Data Segment Register (DS), Stack Segment Register (SS) and Extra Segment Register (ES) respectively. Each of these four registers is 16-bits wide and user accessible—i.e., their contents can be changed by software.

The code segment contains the instruction codes of a program, while data, variables and constants are held in data segment. The stack segment is used to store interrupt and subroutine return addresses. The extra segment contains the destination of data for certain string instructions. Thus 64 KB are available for program storage (in CS) as well as for stack (in SS) while128 KB of space can be utilized for data storage (in DS and ES).One restriction on the base address (starting address) of a segment is that it must reside on a 16-byte address memory—examples being 00000 H, 00010 H or 00020 H, etc.



Non overlapping segments **overlapping segments**

Memory segmentation of 8086

Memory segmentation, as implemented for 8086, gives rise to the following advantages:

- Although the address bus is 20-bits in width, memory segmentation allows one to work with registers having width 16-bits only.

- It allows instruction code, data, stack and portion of program to be more than 64 KB long by using more than one code, data, extra segment and stack segment.
- In a time-shared multitasking environment when the program moves over from one user's program to another, the CPU will simply have to reload the four segment registers with the segment starting addresses assigned to the current user's program.
- User's program (code) and data can be stored separately.
- Because the logical address range is from 0000 H to FFFF H, the same can be loaded at any place in the memory.

Instruction Set of 8086:

There are 117 basic instructions in the instruction set of 8086.The instruction set of 8086 can be divided into the following number of groups, namely:

1. Data copy / Transfer instructions
2. Arithmetic and Logical instructions
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag Manipulation instructions
7. Shift and Rotate instructions
8. String instructions

Data copy / Transfer instructions: The data movement instructions copy values from one location to another. These instructions include **MOV, XCHG, LDS, LEA, LES, PUSH, PUSHF, PUSHFD, POP, POPF, LAHF, AND SAHF.**

MOV The MOV instruction copies a word or a byte of data from source to a destination. The destination can be a register or a memory location. The source can be a register, or memory location or immediate data. MOV instruction does not affect any flags.The mov instruction takes several different forms:

Mov reg, reg1; mov mem, reg; mov reg, mem; mov mem, immediate data; mov reg, immediate data;
mov ax/al, mem; mov mem, ax/al; mov segreg, mem16; mov segreg, reg16; mov mem16, segreg; mov reg16, segreg

The MOV instruction cannot:

1. Set the value of the CS and IP registers.
2. Copy value of one segment register to another segment register (should copy to general register first). MOV CS, DS (Invalid)
3. Copy immediate value to segment register (should copy to general register first). MOV CS, 2000H (Invalid)

Example:

```
ORG 100h
MOV AX, 0B800h;                set AX = B800h
MOV DS, AX;                    copy value of AX to DS.
MOV CL, 'A';              CL = 41h (ASCII code).
```

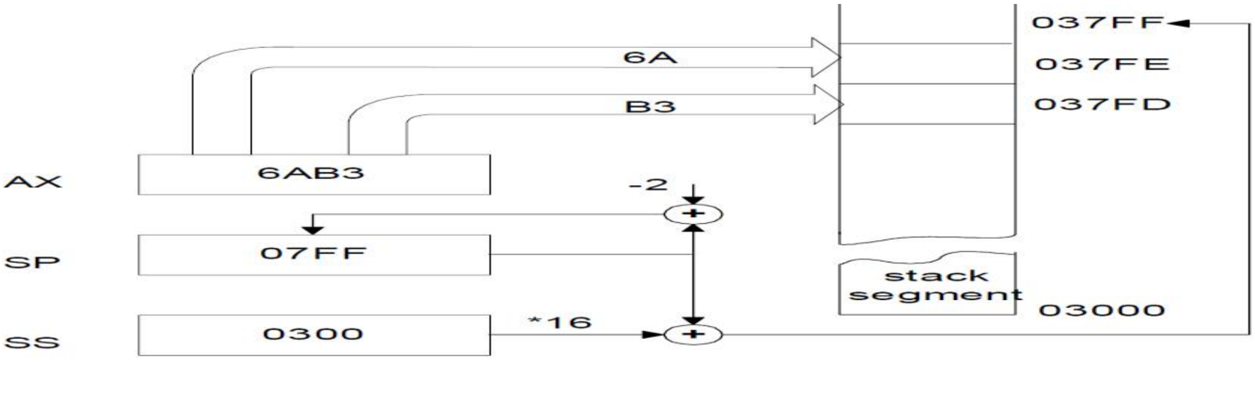
The XCHG Instruction: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them, may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Example: MOV AL, 5; AL = 5

```
 MOV BL, 2; BL = 2
XCHG AL, BL; AL = 2, BL = 5
```

PUSH: Push to stack; this instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

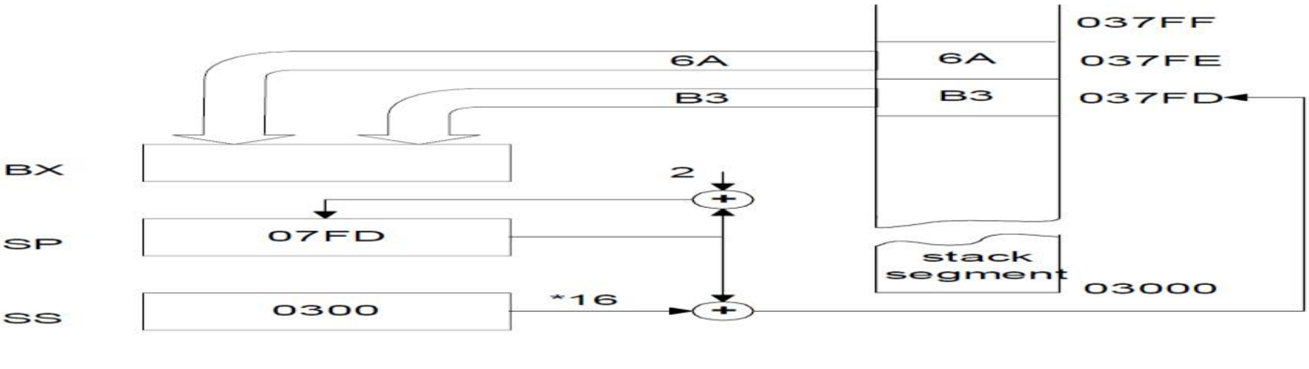
1. PUSH AX
2. PUSH DS
3. PUSH [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.



The effect of PUSH AX instruction

POP: Pop from Stack this instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

1. POP BX
2. POP DS
3. POP [5000H]



The effect of POP BX instruction

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2for each pop operation.

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except overflow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is similar to MOV, MOV is faster than LEA.

LEA cx, [bx+si]; CX (BX+SI) mod 64K **If bx=2f00 H; si=10d0H cx = 3fd0H**

The LDS AND LES instructions:

• LDS and LES load a 16-bit register with offset address retrieved from a memory location then load either DS or ES with a segment address retrieved from memory.

This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers.

• LDS and LES instructions obtain a new far address from memory.

– Offset address appears first, followed by the segment address

• This format is used for storing all 32-bit memory addresses.

• A far address can be stored in memory by the assembler.

LDS BX, DWORD PTR[SI]

BL [SI];

BH [SI+1]

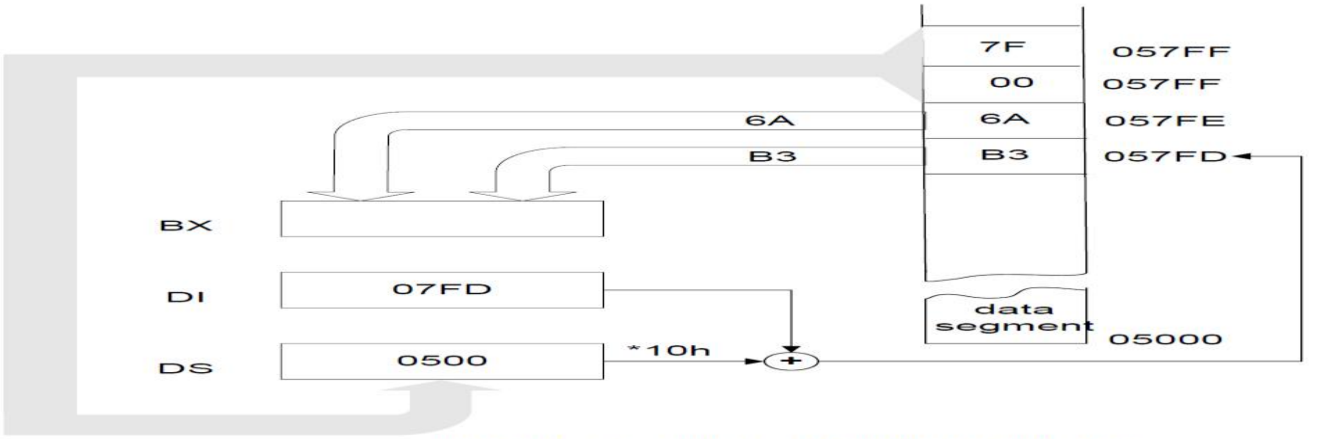
DS [SI+3: SI+2]; in the data segment

LES BX, DWORD PTR[SI]

BL [SI];

BH [SI+1]

ES [SI+3: SI+2]; in the extra segment



The effect of LDS BX,[DI] Instruction

I/O Instructions: The 80x86 supports two I/O instructions: in and out15. They take the forms:

In ax, port

in ax, dx

out port, ax

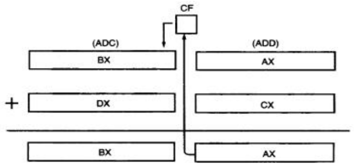
out dx, ax

port is a value between 0 and 255.

The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Arithmetic instructions: These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions.

The ADD and ADC instructions: The add instruction adds the contents of the source operand to the destination operand. For example, **add ax, bx** adds bx to ax leaving the sum in the ax register. **Add computes dest: = dest + source while adc computes dest: = dest + source + C where C represents the value in the carry flag**. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.



Example:

CF=1

BX=25 AX=98

DX=78 CX=94

BX=9E AX=2C

Both instructions affect the flags identically. They set the flags as follows:

- The overflow flag denotes a signed arithmetic overflow.
- The carry flag denotes an unsigned arithmetic overflow.
- The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).
- The zero flag is set if the result of the addition is zero.
- The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.
- The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there is even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote even parity). If there is an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote odd parity).

The INC instruction: The increment instruction adds one to its operand. Except for carry flag, inc sets the flags the same way as Add ax, 1 same as inc ax. The inc operand may be an eight bit, sixteen bit. The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction.

The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. BCD values are decimal integer coded in binary form with one decimal digit (0...9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero (30 ….39).

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add the mas though they were binary numbers and then execute the daa instruction afterwards to correct the results.

Note: These two instructions assume that the add operands were proper decimal or ASCII values. If you add binary (non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC =1) ) then                add al=08 +06; al=0E > 9
al := al + 6                                            al=0E + 06=04
else
ax := ax + 6
end if
ah := ah + 1                                             ah=00+01=01
AuxC := 1 ;Set auxilliary carry
Carry := 1 ; and carry flags.
Else                                                     al=04+03=08, now al<9, so only clear
AuxC := 0 ;Clear auxilliary carry                        ah=0
Carry := 0 ; and carry flags.
endif
al := al and 0Fh
```

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers.

The **daa instruction** functions like aaa except it handles packed BCD values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa’s main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

```
if ( (AL and 0Fh) > 9 or (AuxC = 1)) then        al=24+77=9B, as B>9 add 6 to al
al := al + 6                                     al=9B+06=A1, as higher nibble A>9, add 60
AuxC: = 1 ; Set Auxilliary carry.                to al, al=A1+60=101
End if                                           Note: if higher or lower nibble of AL <9 then
if ( (al > 9Fh) or (Carry = 1)) then             no need to add 6 to AL
al := al + 60h
Carry: = 1; Set carry flag.
End if
```

EXAMPLE:

Assume AL = 0 0 1 1 0 1 0 1, ASCII 5

BL = 0 0 1 1 1 0 0 1, ASCII 9

ADD AL, BL Result: AL= 0 1 1 0 1 1 1 0 = 6EH, which is incorrect BCD

AAA Now AL = 00000100, unpacked BCD 4.

CF = 1 indicates answer is 14 decimal

NOTE: OR AL with 30H to get 34H, the ASCII code for 4. The AAA instruction works only on the AL register. The AAA instruction updates AF and CF, but OF, PF, SF, and ZF are left undefined.

EXAMPLES:

AL = 0101 1001 = 59 BCD; BL = 0011 0101 = 35 BCD

ADD AL, BL AL = 1000 1110 = 8EH

DAA Add 01 10 because 1110 > 9 AL = 1001 0100 = 94 BCD

AL = 1000 1000 = 88 BCD BL = 0100 1001 = 49 BCD

ADD AL, BL AL = 1101 0001, AF=1

DAA Add 0110 because AF =1, AL = 11101 0111 = D7H

1101 > 9 so add 0110 0000

AL = 0011 0111= 37 BCD, CF =1

The DAA instruction updates AF, CF, PF, and ZF. OF is undefined after a DAA instruction.

The SUBTRACTION instructions: SUB, SBB, DEC, AAS, and DAS

The sub instruction computes the value dest: =dest - src. The sbb instruction computes dest: =dest - src - C.

The sub, sbb, and dec instructions affect the flags as follows:

- They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.
- These instructions set the sign flag if the result is negative.
- These instructions set the overflow flag if signed overflow/under flow occurs.
- They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.
- They set the parity flag according to the number of one bits appearing in the result value.
- The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0...9) per byte. This instruction uses the following algorithm:

```
if ( (al and 0Fh) > 9 or AuxC = 1) then
al := al - 6
ah := ah - 1
AuxC: = 1; Set auxilliary carry
Carry: = 1; and carry flags.
else
AuxC: = 0; Clear Auxilliary carry
Carry: = 0; and carry flags.
End if
al := al and 0Fh
```

The das instruction handles the same operation for BCD values, it uses the following Algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC = 1)) then
al := al -6
AuxC = 1
End if
if (al > 9Fh or Carry = 1) then
al := al - 60h
Carry: = 1; Set the Carry flag.
End if
```

EXAMPLE:

ASCII 9-ASCII 5 (9-5)

AL = 00111001 = 39H = ASCII 9

BL = 001 10101 = 35H = ASCII 5

SUB AL, BL Result: AL = 00000100 = BCD 04 and CF = 0

AAS Result: AL = 00000100 = BCD 04 and CF = 0

no borrow required

ASCII 5-ASCII 9 (5-9)

Assume AL = 00110101 = 35H ASCII 5
and BL = 0011 1001 = 39H = ASCII 9
SUB AL, BL Result: AL = 11111100 = - 4 in 2s complement and CF =1
AAS Result: AL = 00000100 = BCD 04 and CF = 1, borrow needed

EXAMPLES:

AL 1000 0110 86 BCD ; BH 0101 0111 57 BCD

SUB AL,BH AL 0010 1111 2FH, CF = 0
DAS Lower nibble of result is 1111, so DAS automatically
Subtracts 0000 0110 to give AL = 00101001 29 BCD
AL 0100 1001 49 BCD BH 0111 0010 72 BCD
SUB AL, BH AL 1101 0111 D7H, CF = 1
DAS Subtracts 0110 0000 (- 60H) because 1101 in upper nibble > 9
AL = 01110111= 77 BCD, CF=1 CF =1 means borrow was needed

The CMP Instruction: The cmp (compare) instruction is identical to the sub instruction with one crucial difference– it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub; the generic form is **cmpdest, src**

Consider the following cmp instruction: cmp ax, bx

This instruction performs the computation ax-bx and sets the flags depending up on the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflows or underflow.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow.This occurs only when ax is less than bx where ax and bx are both unsigned values.

The Multiplication Instructions: MUL, IMUL, and AAM: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX.

The mul instruction, with an **eight bit operand**, multiplies the al register by the operand and **stores the 16 bit result in ax.** So

mul operand	(Unsigned)	MUL BL	i.e. AL * BL; Al=25 * BL=04; AX=00 (AH) 64 (AL)
imul operand	(Signed)	IMUL BL	i.e. AL * BL; AL=09 * BL=-2; AL * 2's comp(BL) AL=09 * BL (0EH) =7E; 2's comp (7e) =-82

The aam (ASCII Adjust after Multiplication) instruction, adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81,the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al: mul bl; al=9, bl=9 al*bl=9*9=51H; AX=00(AH) 51(AL); AAM ; first hexadecimal value is converted to decimal value i.e. 51 to 81; al=81D; second convert packed BCD to unpacked BCD, divide AL content by 10 i.e. 81/10 then AL=01, AH =08; AX = 0801

EXAMPLE:

AL 00000101 unpacked BCD 5

BH 00001001 unpacked BCD 9

MUL BH AL x BH; result in AX

AX = 00000000 00101101 = 002DH

AAM AX = 00000100 00000101 = 0405H, which is unpacked BCD for 45.

If ASCII codes for the result are desired, use next instruction OR AX, 3030H Put 3 in upper nibble of each byte.

AX = 0011 0100 0011 0101 = 3435H, which is ASCII code for 45

The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16division or a 16/8 division. These instructions take the form:

Div reg	For unsigned division
Div mem	
Idiv reg	For signed division
Idiv mem	

The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in .

Note: If an overflow occurs (or you attempt a division by zero) then the80x86 executes an INT 0 (interrupt zero).

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation.It splits apart unpacked binary coded decimal values before an ASCII division operation. The aad instruction is useful forother operations. The algorithm that describes this instruction is

al := ah*10 + al AX=0905H; BL=06; AAD; AX=AH*10+AL=09*10+05=95D; convert decimal to hexadecimal; 95D=5FH; al=5f;
DIV BL; AL/BL=5F/06; AX=05(AH) 0F (AL)

ah := 0

EXAMPLE:

AX = 0607H unpacked BCD for 67 decimal CH = 09H, now adjust to binary

AAD Result: AX = 0043 = 43H = 67 decimal

DIV CH Divide AX by unpacked BCD in CH

Quotient: AL = 07 unpacked BCD Remainder:

AH = 04 unpacked BCD Flags undefined after DIV

NOTE: If an attempt is made to divide by 0, the 8086 will do a type 0 interrupt.

CBW-Convert Signed Byte to Signed Word: This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. The CBW operation must be done beforea signed byte in AL can be divided by another signed byte with the IDIV instruction. CBW affects no flags.

EXAMPLE:

AX = 00000000 10011011 155 decimal

CBW Convert signed byte in AL to signed word in AX

Result: AX = 11111111 10011011 155 decimal

CWD-Convert Signed Word to Signed Double word: CWD copies the sign bit of a word in AX to all the bits of the DX register. In other words it extends the sign of AX into all of DX. The CWD operation must be done before a signed word in AX can be divided by another signed word with the IDIV instruction. CWD affects no flags.

EXAMPLE:

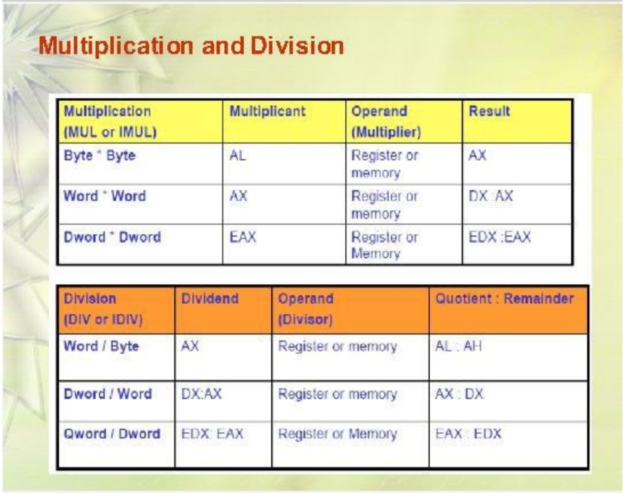
DX = 00000000 00000000

AX = 11110000 11000111 3897 decimal

CWD Convert signed word in AX to signed doubleword in DX:AX

Result DX = 11111111 11111111

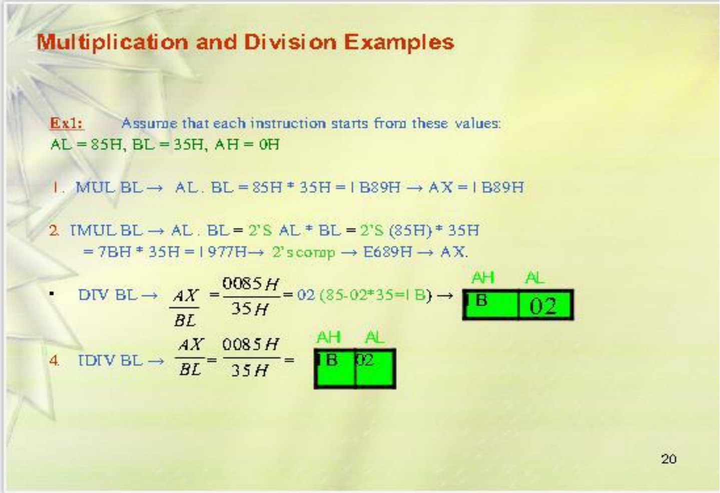
AX = 11110000 11000111 3897 decimal



Multiplication and Division

Multiplication (MUL or IMUL)	Multiplicant	Operand (Multiplier)	Result
Byte * Byte	AL	Register or memory	AX
Word * Word	AX	Register or memory	DX :AX
Dword * Dword	EAX	Register or Memory	EDX :EAX

Division (DIV or IDIV)	Dividend	Operand (Divisor)	Quotient : Remainder
Word / Byte	AX	Register or memory	AL : AH
Dword / Word	DX:AX	Register or memory	AX : DX
Qword / Dword	EDX: EAX	Register or Memory	EAX : EDX



Multiplication and Division Examples

Ex1: Assume that each instruction starts from these values:
AL = 85H, BL = 35H, AH = 0H

1. MUL BL → AL . BL = 85H * 35H = 1B89H → AX = 1B89H

2. IMUL BL → AL . BL = 2'S AL * BL = 2'S (85H) * 35H
= 7BH * 35H = 1977H→ 2'scomp → E689H → AX.

- DIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} = 02$ (85-02*35=1B) →

AH	AL
1B	02

4. IDIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} =$

AH	AL
1B	02

20

Logical, Shift, Rotate and Bit Instructions: The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror,rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar.

The Logical Instructions: AND, OR, XOR, and NOT: The 80x86 logical instructions operate on a bit-by-bit basis. Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

The **AND** instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions. **AND AX, BX**

The **OR** instruction will only set the zero flag if both operands contain zero. **OR AX, BX**

The **XOR** instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form xor reg16, reg16; XOR AX, AX is shorter than the comparable mov reg, 0 instruction.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; Likewise, you can use the or instruction to force certain bits to one in the destination operand;

The Shift Instructions: SHL/SAL, SHR, SAR: The 80x86 supports three different shift instructions (shl and sal are the same instruction): shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right). The general format for a shift instruction is

Shl dest, count sal dest, count shr dest, count sar dest, count

SHL/SAL: These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag.

The shl/sal instruction sets the condition code bits as follows:

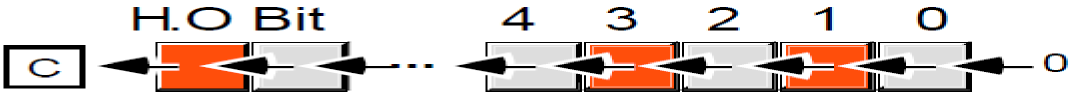
- If the shift count is zero, the shl instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- The overflow flag will contain one if the two H.O. bits were different prior to a single bit shift. The overflow flag is undefined if the shift count is not one.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

shl ah, 4 ;

or al, ah ; Merge in H.O. four bits.

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction).



SHL OPERATION

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

shl ah, 4 ;Move L.O. bits to H.O. position.

and al, 0Fh ;Clear H.O. four bits.

or al, ah ;Merge the bits.

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the **shift left instruction for multiplication by powers of two**:

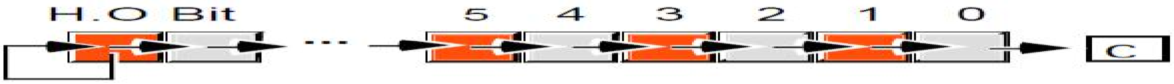
shl ax, 1 ;Equivalent to AX*2

shl ax, 2 ;Equivalent to AX*4

shl ax, 3 ;Equivalent to AX*8

SAR: Thesar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

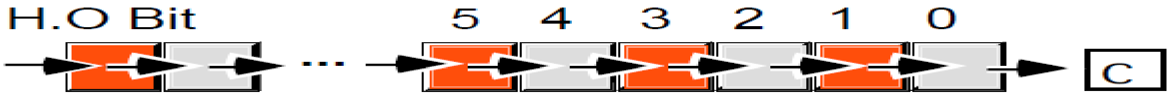


SAR OPERATION

sar ax, 1 ;Signed division by 2
sar ax, 2 ;Signed division by 4
sar ax, 3 ;Signed division by 8
sar ax, 4 ;Signed division by 16
sar ax, 5 ;Signed division by 32
sar ax, 6 ;Signed division by 64
sar ax, 7 ;Signed division by 128
sar ax, 8 ;Signed division by 256

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity.

SHR: The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit



SHR OPERATION

The shift right instruction is especially useful for unpacking data. shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

shr ax, 1 ;Equivalent to AX/2
shr ax, 2 ;Equivalent to AX/4
shr ax, 3 ;Equivalent to AX/8
shr ax, 4 ;Equivalent to AX/16

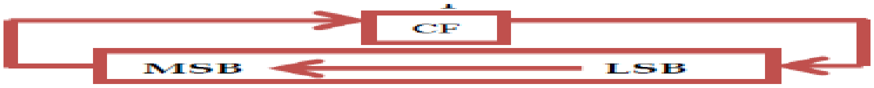
The Rotate Instructions: RCL, RCR, ROL, and ROR

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the operand. They include rcl (rotate through carry left), rcr(rotate through carry right), rol(rotate left), And ror (rotate right). These instructions all take the forms:

```
rcl dest, count       rol dest, count       rcr dest, count       ror dest, count
```

RCL: The rcl (rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right. The rcl instruction sets the flag bits as follows:

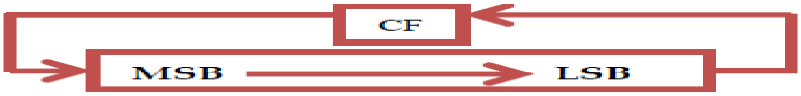
- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
- The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.



RCL OPERATION

RCR: The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit. This instruction sets the flags in a manner analogous to rcl:

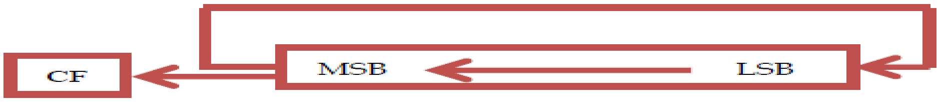
- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.



RCR OPERATION

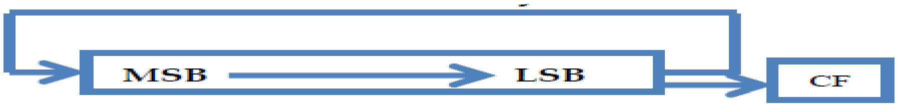
ROL: The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit, rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag. The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction.

Like shl, the rol instruction is often useful for packing and unpacking data.



ROL OPERATION

ROR: The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit.



ROR OPERATION

String Instructions: A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes or words on 8086.**All members of the 80x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS, AND STOS.**

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, we need a source address, a destination address, and a count (the number of string elements to move).The operands for the string instructions include:

- **the SI (source index) register,**
- **the DI (destination index) register,**
- **the CX (count) register,**
- **the AX register, and**
- **the direction flag in the FLAGS register.**

The REP/REPE/REPZ and REPNZ/REPNE Prefixes: The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

Field:

Label repeat mnemonic operand; comment

For MOVS:

Rep movs {operands}

For CMPS:

Repe cmps {operands} repz cmps {operands} repne cmps {operands} repnz cmps {operands}
For SCAS:
Repe scas {operands} repz scas {operands} repnescas {operands} repnzscas {operands}
For STOS:
Rep stos {operands}

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times. Without the repeat prefix, the instruction operates only on a single byte,word, or double word.

If the direction flag is clear, the CPU increments si and di after operating upon each string element. If the direction flag is set, then the 80x86 decrements si and di after processing each string element. The direction flag may be set or cleared using the cld (clear direction flag) and std (setdirection flag) instructions.

The MOVS Instruction: The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es :di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds: si to es: di and decrements the cx register. This process repeats until cx becomes zero. The syntax is:
{REP} MOVSB **{REP} MOVSW**

The CMPS Instruction: The cmps instruction compares two strings. The CPU compares the string referenced by es: di to the string pointed at by ds: si. CX contains the length of the two strings (when using the rep prefix). The syntax is: **{REPE} CMPSB** **{REPE} CMPSW**

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match or length of the string cx=0. The **repe** prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero.

The SCAS Instruction: The scas instruction, by itself, compares the value in the accumulator (al or ax) against the value pointed at by es:di and then increments (or decrements) di by one or two. The CPU sets the flags according to the result of the comparison. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator. The scas instruction takes the following forms: **{REPNE} SCASB** **{REPNE} SCASW**

The STOS Instruction: The stos instruction stores the value in the accumulator at the location specified by es: di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Its primary use is to initialize arrays and strings to a constant value. **{REP} STOSB**
{REP} STOSW

The LODS Instruction: The lods instruction copies the byte or word pointed at by ds:si into the al or ax register, after which it increments or decrements the si register by one or two.**{REP} LODSB**
{REP} LODSW

Flag Manipulation and Processor Control Instructions: These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions.

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

CLC - Clear carry flag **CMC - Complement carry flag** **STC - Set carry flag**
CLD - Clear direction flag **STD - Set direction flag** **CLI - Clear interrupt flag**

STI - Set interrupt flag

These instructions modify the carry (CF), direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto increment or auto decrement modes.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

WAIT - Wait for Test input pin to go low HLT - Halt the processor NOP - No operation ESC - Escape to external device like NDP (numeric co-processor) LOCK - Bus lock instruction prefix.

After executing the **HLT instruction**, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it.

When **NOP instruction** is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP byone. It then continues with further execution after 4 clock cycles.

ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices.

The **LOCK prefix** may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems.

The **WAIT instruction** when executed holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Program Flow Control Instructions: The control transfer instructions are used to transfer the control from one memory location to another memory location. In 8086 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions.

Unconditional Jumps: The jmp (jump) instruction unconditionally transfers control to another point in the program. Intra segment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

JMP Address

P
a
r
t

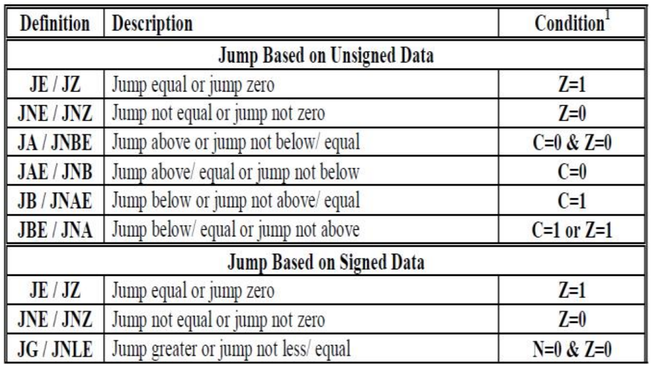
1

J
M
P

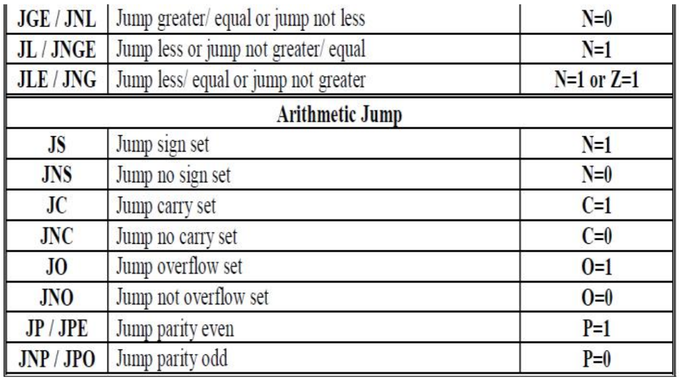
A
A

Unconditional jump **Conditional jump**

Conditional Jump: The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if…..then statement. The conditional jumps test one or more bits in the status register to see if they match some particular pattern. If the pattern matches, control transfers to the target location. If the condition fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow and zero flags.



Definition	Description	Condition[1]
	Jump Based on Unsigned Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JA / JNBE	Jump above or jump not below/ equal	C=0 & Z=0
JAE / JNB	Jump above/ equal or jump not below	C=0
JB / JNAE	Jump below or jump not above/ equal	C=1
JBE / JNA	Jump below/ equal or jump not above	C=1 or Z=1
	Jump Based on Signed Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JG / JNLE	Jump greater or jump not less/ equal	N=0 & Z=0



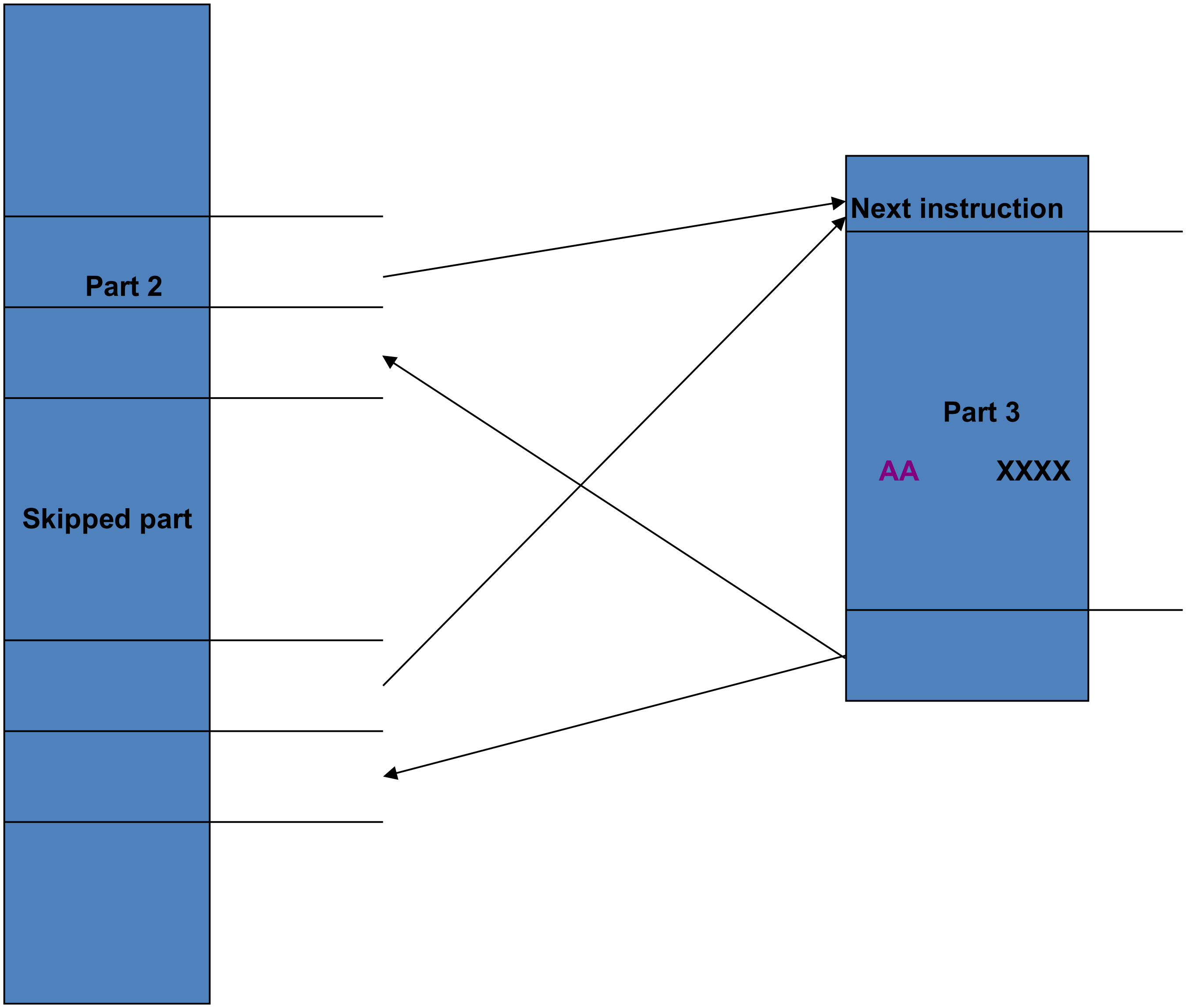
JGE / JNL	Jump greater/ equal or jump not less	N=0
JL / JNGE	Jump less or jump not greater/ equal	N=1
JLE / JNG	Jump less/ equal or jump not greater	N=1 or Z=1
	Arithmetic Jump	
JS	Jump sign set	N=1
JNS	Jump no sign set	N=0
JC	Jump carry set	C=1
JNC	Jump no carry set	C=0
JO	Jump overflow set	O=1
JNO	Jump not overflow set	O=0
JP / JPE	Jump parity even	P=1
JNP / JPO	Jump parity odd	P=0

Loop Instruction:

- These instructions are used to repeat a set of instructions several times.
- Format: LOOP Short-Label
- Operation: (CX) 🡐 (CX)-1
- Jump is initialized to location defined by short label if CX≠0. Otherwise, execute next sequential instruction.
- Instruction LOOP works with respect to contents of CX. CX must be preloaded with a count that represents the number of times the loop is to be repeat.

- Whenever the loop is executed, contents at CX are first decremented then checked to determine if they are equal to zero.
- If CX=0, loop is complete and the instruction following loop is executed.
- If CX ≠ 0, content return to the instruction at the label specified in the loop instruction.
- **LOOP AGAIN is <u>almost same</u> as: DEC CX, JNZ AGAIN**

SUBROUTINE & SUBROUTINE HANDILING INSTRUCTIONS: CALL, RET



- A subroutine is a special segment of program that can be called for execution from any point in a program.
- An assembly language subroutine is also referred to as a “procedure”.
- Whenever we need the subroutine, a single instruction is inserted in to the main body of the program to call subroutine.
- Transfers the flow of the program to the procedure.

- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
- To branch a subroutine the value in the IP or CS and IP must be modified.
- After execution, we want to return the control to the instruction that immediately follows the one called the subroutine i.e., the original value of IP or CS and IP must be preserved.
- Execution of the instruction causes the contents of IP to be saved on the stack. (this time (SP) ← (SP) -2)
- A new 16-bit (near-proc, mem16, reg16 i.e., Intra Segment) value which is specified by the instructions operand is loaded into IP.
- Examples: CALL 1234H
  CALL BX
  CALL [BX]

Return Instruction: RET instruction removes an address from the stack so the program returns to the instruction following the CALL

- Every subroutine must end by executing an instruction that returns control to the main program. This is the return (RET) instruction.
- By execution the value of IP or IP and CS that were saved in the stack to be returned back to their corresponding registers. (this time (SP) ← (SP)+2)

MACROS: The macro directive allows the programmer to write a named block of source statements, then use that name in the source file to represent the group of statements. During the assembly phase, the assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.

Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if used repeatably. Procedures or subroutines take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower. In summary, the advantages and disadvantages of macros are,

Advantages

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages

In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Modular Programming: Instead of writing a large program in a single unit, it is better to write small programs—which are parts of the large program. Such small programs are called program modules or simply modules. Each such module can be separately written, tested and debugged. Once the debugging of the small programs is over, they can be linked together. Such methodology of developing a large program by linking the modules is called modular programming.

Assembler Directives:

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ, assume and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, to generate address.

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and byte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

ASSUME: The ASSUME directive tell the assembler the name of the logical segment it should use for a specified segment. Ex: ASSUME CS: Code, DS: Data, SS: Stack; or ASSUME CS: Code

Data Directives: The directives DB, DW, DD, DR and DT are used to (a) define different types of variables or (b) to set aside one or more storage locations in memory-depending on the data type:

DB — Define Byte DW — Define Word DD — Define Double word
DQ — Define Quad word DT — Define Ten Bytes

The **DB directive** is used to declare a byte-type variable or to set aside one or more storage locations of type byte in memory (Define Byte)

Example: Temp DB 42H; Temp is a variable allotted 1byte of memory location assigned with data 42H

The **DW directive** is used to declare a variable of type word or to reserve memory locations which can be accessed as type double word (Define word)

Example: N2 DW 427AH; N2 variable is initialized with value 427AH when it is loaded into memory to run.

The **DD directive** is used to declare a variable of type double word or to reserve memory locations which can be accessed as type double word (Define double word)

Example: Big DD 2456756CH; Big variable is initialized with 4 bytes

The **DQ directive** is used to tell the assembler to declare a variable 4 words in length or to reverse 4 words of storage in memory (Define Quad word)

Example: Big DQ 2456756C88464567H; Big variable is initialized with 4 words (8 bytes)

The **DT directive** is used to tell the assembler to declare a variable 10 bytes in length or to reverse 10bytes of storage in memory (Define Ten bytes)

Example: Packed BCD DT 11223344556677889900H; 10 byte data is initialized to variable packed BCD

DUP: This directive operator is used to initialize several locations and to assign values to these locations. Its format is: Name Data-Type Num DUP (value)

Example: TABLE DB 20 DUP (0); Reserve an array of 20 bytes of memory and initialize all 20 bytes with 0. Array is named TABLE

END: The **END** directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an end directive.

The **ENDP** directive is used with the name of the procedure to indicate the end of a procedure to the assembler.

```
SQUARE NUM PROC
….
….
SQUARE NUM ENDP
```

The **ENDS** directive is used with the name of the segment to indicate the end of a segment to the assembler.

```
CODE SEGMENT
…
…
CODE ENDS
```

EQU: The **EQU** directive is used to give a name to some value or to a symbol. Each time assembler finds the name in the program it will replace the name with the value.

FACTOR EQU 03H; This statement should be written at the start

ADD AL, FACTOR; The assembler converts this instruction as ADD AL, 03H

EVEN: The **EVEN** directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word starts at an odd address, 8086 will take 2 bus cycles to get the 2 byte of the word. *"A series of words can read much more quickly if they are at even address"*.

```
DATA HERE SEGMENT      ; Location counter will point to 0009H after assembler reads next
statement
SALES DB 9 DUP (?)     ; Declare an array of 9 bytes
EVEN                   ; Increment location counter to 000AH
RECORD DW 100 DUP (?)  ; Array of 100 words starting on even address for quicker read
DATA HERE ENDS         ;
```

GLOBAL: This **GLOBAL** directive can be used in place of PUBLIC directive or in place of an EXTRN directive. The GOLBAL directive is used to make the symbol available to other modules.

PUBLIC: The **PUBLIC** directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive.

EXTRN: This **EXTRN** directive is used to tell the assembler that the names or labels following the directive are in some other assembly module.

GROUP: This **GROUP** directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment.

Example: SMALL SYSTEM GROUP CODE, DATA, STACK

ASSUME CS: SMALL SYSTEM, DS: SMALL SYSTEM, SS: SMALL SYSTEM

OFFSET—Is an operator which tells the assembler to determine the offset or the displacement of a named data item (variable) or procedure from start of the segment which contains it. This operator is used to load the offset of a variable into a register so that the variable can be accessed with one of the indexed addressing modes. MOV AL, OFFSET N1

ORG – This **ORG** directive allows to set the location counter to a desired value at any point in the program. The statement ORG 100H tells the assembler to set the location counter to 0100H.

PROCEDURE: A PROC directive is used to define a label and to delineate a sequence of instructions that are usually interpreted to be a subroutine, that is, CALLed either from within the same physical segment (near) or from another physical segment (far).

Syntax:

```
name PROC [type]                    P1 PROC NEAR
                                    MOV AX, 1 5
                                    ADD OX, AX
…..                                 ENDP
name ENDP
```

Labels: A label, a symbolic name for a particular location in an instruction sequence, maybe defined in one of three ways. The first way is the most common. The format is shown below: **label: [instruction]**

where "label" is a unique ASM86 identifier and "instruction" is an*8086/8087/8088* instruction. This label will have the following attributes:

1. Segment-the current segment being assembled.
2. Offset-the current value of the location counter.
3. Type-will be NEAR.

An example of this form of label definition is: ALAB: MOV AX, COUNT

Introduction to 8051 MicroContoller:

To make a complete microcomputer system, only microprocessor is not sufficient. It is necessary to add other peripherals such as ROM, RAM, decoders, drivers, number of I/O devices to make a complete microcomputer system. In addition, special purpose devices, such as interrupt controller, programmable timers, programmable I/O devices, DMA controllers may be added to improve the capability and performance and flexibility of a microcomputer system.

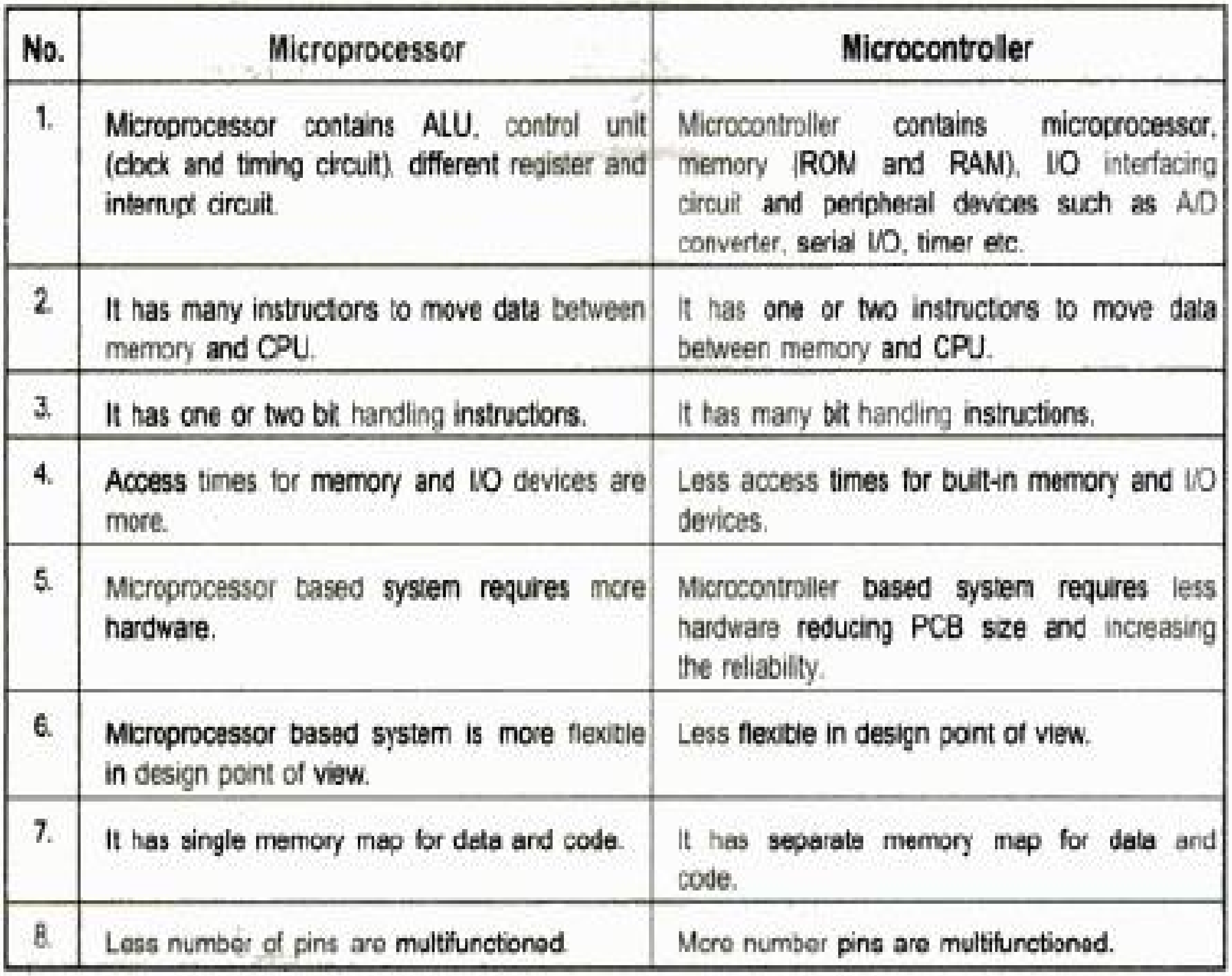
The key feature for microprocessor based design is that it has more flexibility to configure a system as large system or small system by adding suitable peripherals.

On the other hand, the microcontroller incorporates all the features that are found in microprocessor. The microcontroller has built-in ROM, RAM, parallel I/O, serial I/O, counters and a clock circuit. It has on-chip peripheral devices which makes it possible to have single microcomputer system.

Advantages of built-in peripherals:

Built-in peripherals have smaller access times hence speed is more.
Hardware reduces due to single chip microcomputer system.
Less hardware reduces PCB size and increases reliability of the system.

Comparison between Microprocessors and Microcontrollers:

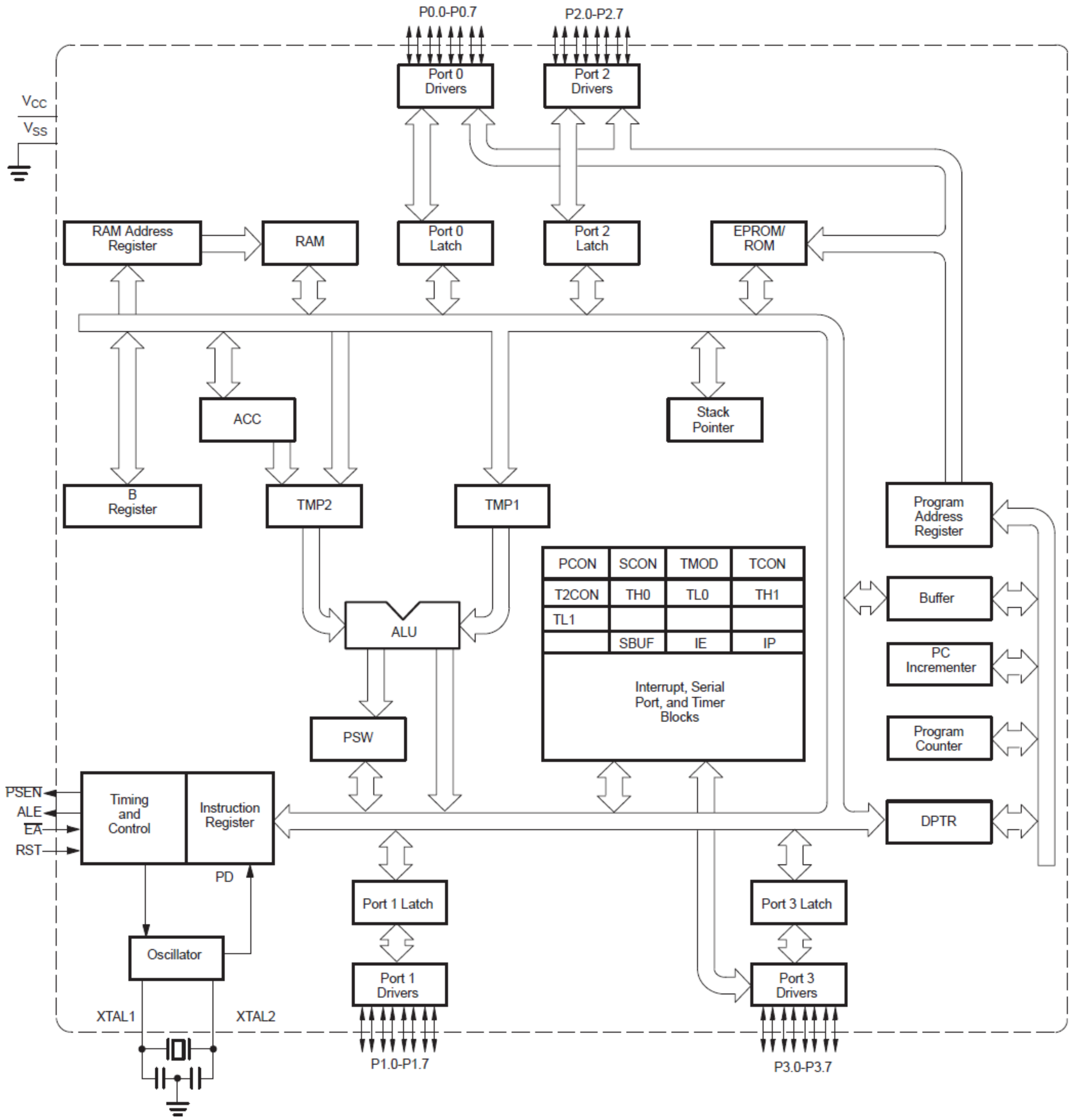


No.	Microprocessor	Microcontroller
1.	Microprocessor contains ALU, control unit (clock and timing circuit), different register and interrupt circuit.	Microcontroller contains microprocessor, memory (ROM and RAM), I/O interfacing circuit and peripheral devices such as A/D converter, serial I/O, timer etc.
2.	It has many instructions to move data between memory and CPU.	It has one or two instructions to move data between memory and CPU.
3.	It has one or two bit handling instructions.	It has many bit handling instructions.
4.	Access times for memory and I/O devices are more.	Less access times for built-in memory and I/O devices.
5.	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing the reliability.
6.	Microprocessor based system is more flexible in design point of view.	Less flexible in design point of view.
7.	It has single memory map for data and code.	It has separate memory map for data and code.
8.	Less number of pins are multifunctioned.	More number pins are multifunctioned.

Features of 8051:

- 4KB on-chip program memory (ROM/EPROM).
- 128 bytes on-chip data memory.
- Four register banks.
- 64KB each program and external RAM addressability.
- One microsecond instruction cycle with 12MHz crystal.
- 32 bidirectional I/O lines organized as four 8-bit ports.
- Multiple modes, high-speed programmable serial port (UART).
- 16-bit Timers/Counters.
- Direct byte and bit addressability.

Block Diagram of 8051:



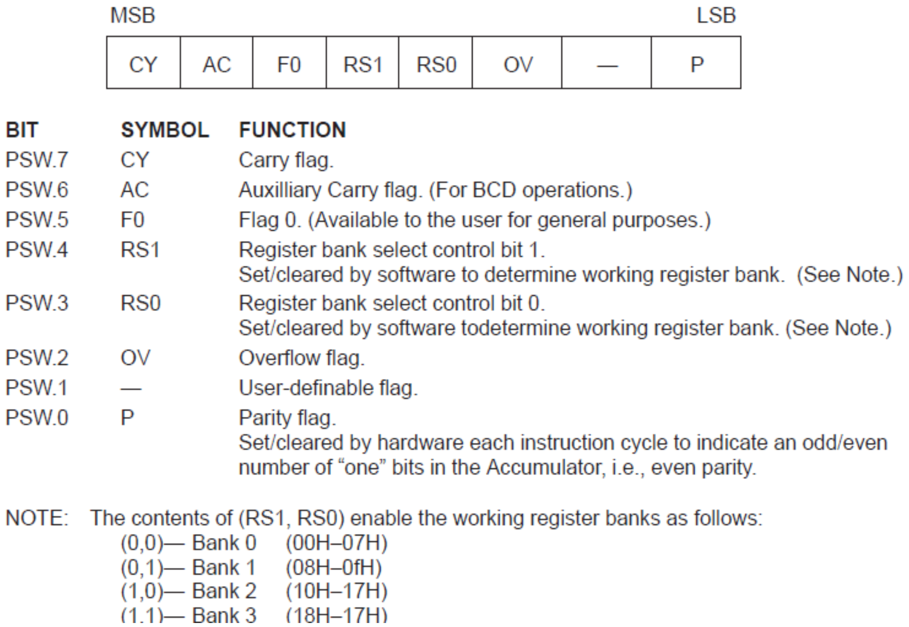
Accumulator: The Accumulator, as it's name suggests, is used as a general register to accumulate the results of a large number of instructions. It can hold an 8-bit (1-byte) value.

'B' Register: The "B" register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value. The "B" register is only used by two 8051 instructions: MUL AB and DIV AB.

Aside from the MUL and DIV an instruction, the "B" register is often used as yet another temporary storage register much like a ninth "R" register.

Program Status Word

The PSW register contains program status information. It is a 8-bit flag register, out of 8-bits 6 bits are used and 2 bits are reserved. Out of 6 bits 4 bits are conditional bits and 2 bits are used for selecting register bank.



MSB LSB

CY	AC	F0	RS1	RS0	OV	—	P

BIT	SYMBOL	FUNCTION
PSW.7	CY	Carry flag.
PSW.6	AC	Auxilliary Carry flag. (For BCD operations.)
PSW.5	F0	Flag 0. (Available to the user for general purposes.)
PSW.4	RS1	Register bank select control bit 1. Set/cleared by software to determine working register bank. (See Note.)
PSW.3	RS0	Register bank select control bit 0. Set/cleared by software todetermine working register bank. (See Note.)
PSW.2	OV	Overflow flag.
PSW.1	—	User-definable flag.
PSW.0	P	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of "one" bits in the Accumulator, i.e., even parity.

NOTE: The contents of (RS1, RS0) enable the working register banks as follows:

(0,0)—	Bank 0	(00H–07H)
(0,1)—	Bank 1	(08H–0fH)
(1,0)—	Bank 2	(10H–17H)
(1,1)—	Bank 3	(18H–17H)

Stack Pointer

The Stack Pointer register is 8 bits wide. It is incremented before data is stored during PUSH and CALL executions. While the stack may reside anywhere in on-chip RAM, the Stack Pointer is initialized to 07H after a reset. This causes the stack to begin at locations 08H.

Data Pointer

The Data Pointer (DPTR) consists of a high byte (DPH) and a low byte (DPL). Its intended function is to hold a 16-bit address. It may be manipulated as a 16-bit register or as two independent 8-bit registers.

Program Counter

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized PC always starts at 0000h and is incremented each time an instruction is executed. It is important to note that PC isn't always incremented by one. Since some instructions require 2 or 3 bytes the PC will be incremented by 2 or 3 in these cases.

The Program Counter is special in that there is no way to directly modify its value. That is to say, we can't do something like PC=2430h. On the other hand, if we execute LJMP 2430h you've effectively accomplished the same thing.

Ports 0 to 3

P0, P1, P2, and P3 are the SFR latches of Ports 0, 1, 2, and 3, respectively. Writing a one to a bit of a port SFR (P0, P1, P2, or P3) causes the corresponding port output pin to switch high. Writing a zero causes the port output pin to switch low. When used as an input, the external state of a port pin will be held in the port SFR (i.e., if the external state of a pin is low, the corresponding port SFR bit will contain a 0; if it is high, the bit will contain a 1).

Serial Data Buffer

The Serial Buffer is actually two separate registers, a transmit buffer and a receive buffer. When data is moved to SBUF, it goes to the transmit buffer and is held for serial transmission. (Moving a byte to SBUF is what initiates the transmission.) When data is moved from SBUF, it comes from the receive buffer.

Timer Registers Basic to 80C51

Register pairs (TH0, TL0), and (TH1, TL1) are the 16-bit Counting registers for Timer/Counters 0 and 1, respectively.

Control Register for the 80C51

Special Function Registers IP, IE, TMOD, TCON, SCON, and PCON contain control and status bits for the interrupt system, the Timer/Counters, and the serial port.

Register Banks

The 8051 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7). These registers are generally used to assist in manipulating values and moving data from one memory location to another.

PSEN (Program Store Enable)

The 8051 has four dedicated bus control signals. It is a control signal that enables external program (code) memory**.** It usually connects to an EPROM's Output Enable (OE) pin to permit reading of program bytes. The PSEN signal pulses low during the fetch stage of an instruction. When executing a program from internal ROM (8051/8052), PSEN remains in the inactive (high) state.

ALE (Address Latch Enable)

The 8051 similarly uses ALE for demultiplexing the address and data bus. When Port 0 is used in its alternate mode—as the data bus and the low-byte of the address bus—ALE is the signal that latches the address into an external register during the first half of a memory cycle.

EA (External Access)

The EA input signal is generally tied high (+5 V) or low (ground). If high, the 8051 executes programs from internal ROM when executing in the lower 4K of memory. If low, programs execute from external memory only (and PSEN pulses low accordingly).

RST (Reset)

The RST input is the master reset for the 8051. When this signal is brought high for at least two machine cycles, the 8051 internal registers are loaded with appropriate values for an orderly system start-up.

On-chip Oscillator Inputs

The 8051 features an on-chip oscillator. The nominal crystal frequency is 12 MHz for most ICs in the MCS-51™ family.

Memory Organization

Most microprocessors implement a shared memory space for data and programs. This is reasonable, since programs are usually stored on a disk and loaded into RAM for execution; thus both the data and programs reside in the system RAM. Microcontrollers have limited memory, and there is no disk drive or disk operating system. The control program must reside in. For this reason, the 8051 implements a separate memory space for programs (code) and data. Both the code and data may be internal; however, both expand using external components to a maximum of 64K code memory and 64K data memory.

The internal memory consists of on-chip ROM (8051/8052 only) and on-chip data RAM. ***The on-chip RAM contains a rich arrangement of general-purpose storage, bit-addressable storage, register banks, and special function registers.***

The internal memory space is divided between register banks (00H-1FH), bit-addressable RAM (20H-2FH), general-purpose RAM (30H-7FH), and special function registers (80H-FFH).

Any location in the general-purpose RAM can be accessed freely using the direct or indirect addressing modes.

Bit-addressable RAM

The 8051 contains 210 bit-addressable locations, of which 128 are at byte addresses 20H through 2FH, and the rest are in the special function registers.

The idea of individually accessing bits through software is a powerful feature of most microcontrollers. Bits can be set, cleared, ANDed, ORed, etc., with a single instruction.

Most microprocessors require a read-modify-write sequence of instructions to achieve the same effect. Furthermore, the 8051 I/O ports are bit-addressable, simplifying the software interface to single-bit inputs and outputs.

There are 128 general-purpose bit-addressable locations at byte address 20H through 2FH (8 bits/byte X 16 bytes = 128 bits).

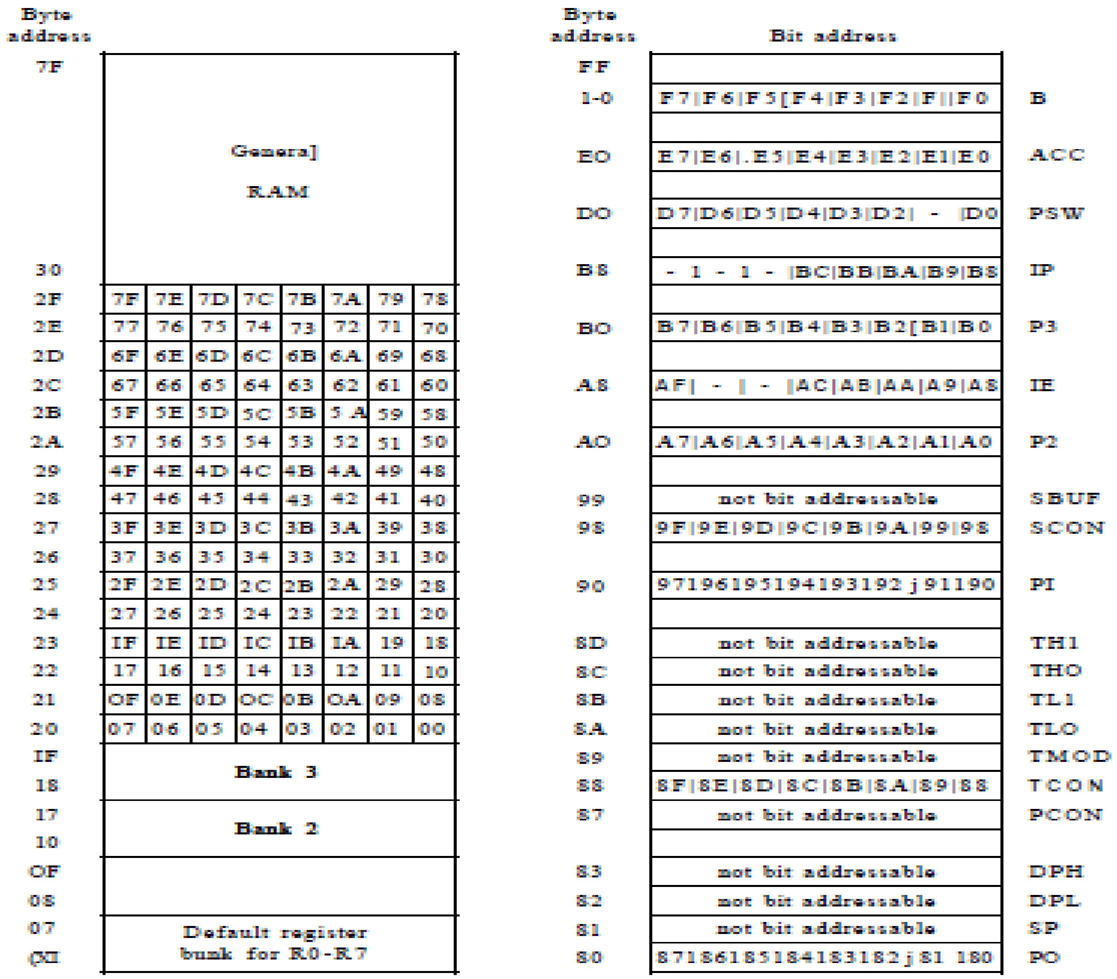
Register Banks

The bottom 32 locations of internal memory contain the register banks. The 8051 instruction set supports 8 registers, R0 through R7, and by default (after a system reset) these registers are at addresses OOH-07H.

Instructions using registers R0 to R7 are shorter and faster than the equivalent instructions using direct addressing. Data values used frequently should use one of these registers.

Special Function Registers

The 8051 internal registers are configured as part of the on-chip RAM; therefore, each register also has an address. This is reasonable for the 8051, since it has so many registers. As well as R0 to R7, there are 21 special function registers (SFRs) at the top of internal RAM, from addresses 80H to FFH.



Byte address								
7F	Genera] RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5 A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	IF	IE	ID	IC	IB	IA	19	18
22	17	16	15	14	13	12	11	10
21	OF	0E	0D	OC	0B	OA	09	08
20	07	06	05	04	03	02	01	00
IF	Bank 3							
18								
17	Bank 2							
10								
OF								
08								
07	Default register bunk for R0-R7							
(XI								

Byte address	Bit address	
FF		
1-0	F7|F6|F5[F4|F3|F2|F||F0	B
EO	E7|E6|.E5|E4|E3|E2|E1|E0	ACC
DO	D7|D6|D5|D4|D3|D2| - |D0	PSW
B8	- 1 - 1 - |BC|BB|BA|B9|B8	IP
BO	B7|B6|B5|B4|B3|B2[B1|B0	P3
A8	AF| - | - |AC|AB|AA|A9|A8	IE
AO	A7|A6|A5|A4|A3|A2|A1|A0	P2
99	not bit addressable	SBUF
98	9F|9E|9D|9C|9B|9A|99|98	SCON
90	9719619519419319 2 j 91190	PI
8D	not bit addressable	TH1
8C	not bit addressable	THO
8B	not bit addressable	TL1
8A	not bit addressable	TLO
89	not bit addressable	TMOD
88	8F|8E|8D|8C|8B|8A|89|88	TCON
87	not bit addressable	PCON
83	not bit addressable	DPH
82	not bit addressable	DPL
81	not bit addressable	SP
80	8718618518418318 2 j 81 180	PO

SPKCIAL FUNCTION REGISTERS

EXTERNAL MEMORY

The MCS-51 architecture provides expansion in the form of a 64K external code memory space and a 64K external data memory space. Extra ROM and RAM can be added as needed. Peripheral interface ICs can also be added to expand the I/O capability. These *become* part of the external data memory space using memory-mapped I/O.

When external memory is used, Port 0 is unavailable as an I/O port. It becomes a multiplexed address (A0-A7) and data (D0-D7) bus, with ALE latching the low-byte of the address at the beginning of each external memory cycle. Port 2 is usually (but not always) employed for the high-byte of the address bus.

Addressing Modes of 8051:

In this section, we will see different addressing modes of the 8051 microcontrollers. In 8051 there are 1-byte, 2-byte instructions and very few 3-byte instructions are present. The opcodes are 8-bit long. As the opcodes are 8-bit data, there are 256 possibilities. Among 256, 255 opcodes are implemented.

The clock frequency is12MHz, so 64 instruction types are executed in just 1 µs, and rest are just 2 µs. The Multiplication and Division operations take 4 µsto to execute.

In 8051 There are six types of addressing modes.

- Immediate AddressingMode
- Register AddressingMode
- Direct AddressingMode
- Register IndirectAddressing Mode
- Indexed AddressingMode
- Implied AddressingMode

Immediate addressing mode

In this Immediate Addressing Mode, the data is provided in the instruction itself. The data is provided immediately after the opcode. These are some examples of Immediate Addressing Mode.

```
MOVA, #0AFH;
MOVR3, #45H;
MOVDPTR, #FE00H;
```

In these instructions, the # symbol is used for immediate data. In the last instruction, there is DPTR. The DPTR stands for Data Pointer. Using this, it points the external data memory location. In the first instruction, the immediate data is AFH, but one 0 is added at the beginning. So when the data is starting with A to F, the data should be preceded by 0.

Register addressing mode

In the register addressing mode the source or destination data should be present in a register (R0 to R7). These are some examples of RegisterAddressing Mode.

```
MOVA, R5;
MOVR2, #45H;
MOVR0, A;
```

In 8051, there is no instruction like **MOVR5, R7**. But we can get the same result by using this i

nstruction **MOV R5, 07H**, or by using **MOV 05H, R7**. But this two instruction will work when the selected register bank is **RB0**. To use another register bank and to get the same effect, we have to add the starting address of that register bank with the register number. For an example, if the RB2 is selected, and we want to access R5, then the address will be (10H + 05H = 15H), so the instruction will look like this **MOV 15H, R7**. Here 10H is the starting address of Register Bank 2.

Direct Addressing Mode

In the Direct Addressing Mode, the source or destination address is specified by using 8- bit data in the instruction. Only the internal data memory can be used in this mode. Here some of the examples of direct Addressing Mode.

```
MOV80H,  R6;
MOVR2,  45H;
MOVR0, 05H;
```

The first instruction will send the content of registerR6 to port P0 (Address of Port 0 is 80H). The second one is forgetting content from 45H to R2. The third one is used to get data from Register R5 (When register bank RB0 is selected) to register R5.

Register indirect addressing Mode

In this mode, the source or destination address is given in the register. By using register indirect addressing mode, the internal or external addresses can be accessed. The R0 and R1 are used for 8-bit addresses, and DPTR is used for 16-bit addresses, no other registers can be used for addressing purposes. Let us see some examples of this mode.

```
MOV0E5H, @R0; MOV@R1,
80H
```

In the instructions, the @ symbol is used for register indirect addressing. In the first instruction, it is showing that theR0 register is used. If the content of R0 is 40H, then that instruction will take the data which is located at location 40H of the internal RAM. In the second one, if the content of R1 is 30H, then it indicates that the content of port P0 will be stored at location 30H in the internal RAM.

```
MOVXA, @R1;
MOV@DPTR, A;
```

In these two instructions, the X in MOVX indicates the external data memory. The external data memory can only be accessed in register indirect mode. In the first instruction if the R0 is holding 40H, then A will get the content of external RAM location40H. And in the second one, the content of A is overwritten in the location pointed by DPTR.

Indexed addressing mode

In the indexed addressing mode, the source memory can only be accessed from program memory only. The destination operand is always the register A. These are some examples of Indexed addressing mode.

```
MOVCA, @A+PC;
MOVCA, @A+DPTR;
```

The C in MOVC instruction refers to code byte. For the first instruction, let us consider A holds 30H. And the PC value is1125H. The contents of program memory location 1155H (30H + 1125H) are moved to register A.

Implied Addressing Mode

In the implied addressing mode, there will be a single operand. These types of instruction can work on specific registers only. These types of instructions are also known as register specific instruction. Here are some examples of Implied Addressing Mode.

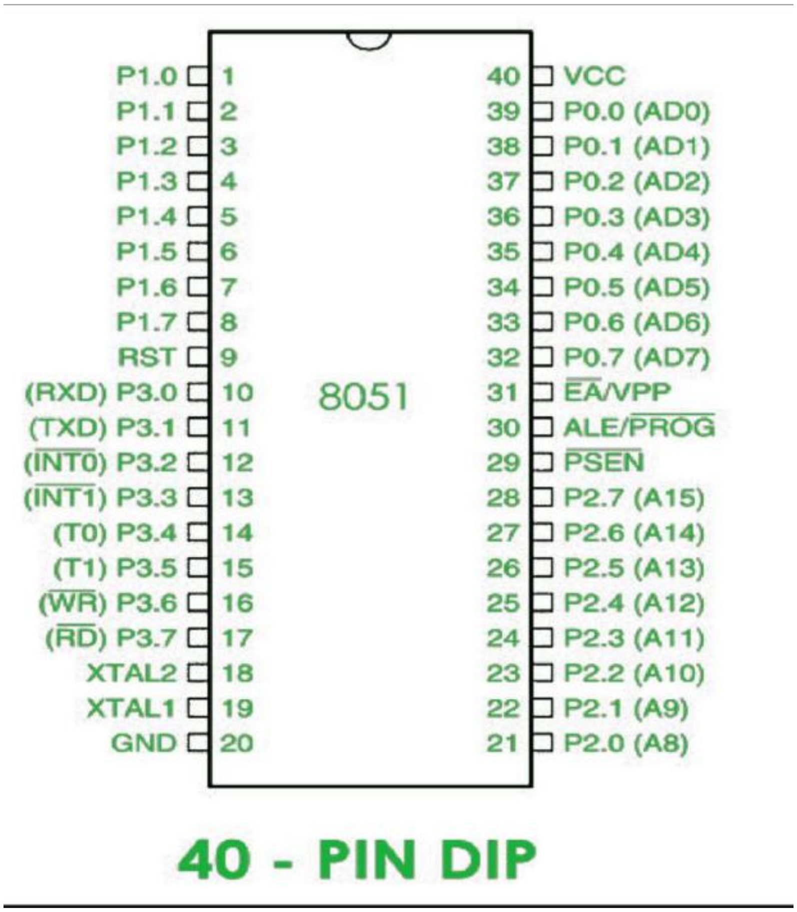
```
RLA;
SWAPA;
```

These are 1- byte instruction. The first one is used to rotate the A register content to the Left. The second one is used to swap the nibbles in A.

Pin Diagram of 8051:

8051 microcontroller is a 40 pin Dual Inline Package (DIP). These 40 pins serve different functions like read, write, I/O operations, interrupts etc. 8051 has four I/O ports wherein each port has 8 pins which can be configured as input or output depending upon the logic state of the pins. Therefore, 32 out of these 40 pins are dedicated to I/O ports. The rest of the pins are dedicated to VCC, GND, XTAL1, XTAL2, RST, ALE, EA’ and PSEN’.

Pin diagram of 8051 microprocessor is as given below :



Description of the Pins :

- Pin 1 to Pin 8 (Port 1) –

Pin 1 to Pin 8 are assigned to Port 1 for simple I/O operations. They can be configured as input or output pins depending on the logic control i.e. if logic zero

(0) is applied to the I/O port it will act as an output pin and if logic one (1) is applied the pin will act as an input pin. These pins are also referred to as P1.0 to P1.7 (where P1 indicates that it is a pin in port 1 and the number after '.' tells the pin number i.e. 0 indicates first pin of the port. So, P1.0 means first pin of port 1, P1.1 means second pin of the port 1 and so on). These pins are bidirectional pins.

- Pin 9 (RST) –

  Reset pin. It is an active-high, input pin. Therefore if the RST pin is high for a minimum of 2 machine cycles, the microcontroller will reset i.e. it will close and terminate all activities. It is often referred as "power-on-reset" pin because it is used to reset the microcontroller to it's initial values when power is on (high).

- Pin 10 to Pin 17 (Port 3) –

  Pin 10 to pin 17 are port 3 pins which are also referred to as P3.0 to P3.7. These pins are similar to port 1 and can be used as universal input or output pins. These pins are bidirectional pins.

- These pins also have some additional functions which are as follows:

1) **P3.0 (RXD):**

10th pin is RXD (serial data receive pin) which is for serial input. Through this input signal microcontroller receives data for serial communication.

2) **P3.1 (TXD):**

11th pin is TXD (serial data transmit pin) which is serial output pin. Through this output signal microcontroller transmits data for serial communication.

3) **P3.2 and P3.3 (INT0′, INT1′) :**

12th and 13th pins are for External Hardware Interrupt 0 and Interrupt 1 respectively. When this interrupt is activated(i.e. when it is low), 8051 gets interrupted in whatever it is doing and jumps to the vector value of the interrupt (0003H for INT0 and 0013H for INT1) and starts performing Interrupt Service Routine (ISR) from that vector location.

4) **P3.4 and P3.5 (T0 and T1) :**

14th and 15th pin are for Timer 0 and Timer 1 external input. They can be connected with 16 bit timer/counter.

5) **P3.6 (WR') :**

16th pin is for external memory write i.e. writing data to the external memory.

6) **P3.7 (RD') :**

17th pin is for external memory read i.e. reading data from external memory.

- Pin 18 and Pin 19 (XTAL2 And XTAL1) –

These pins are connected to an external oscillator which is generally a quartz crystal oscillator. They are used to provide an external clock frequency of 4MHz to 30MHz.

- **Pin 20 (GND) :**

This pin is connected to the ground. It has to be provided with 0V power supply. Hence it is connected to the negative terminal of the power supply.

- **Pin 21 to Pin 28 (Port 2) :**

Pin 21 to pin 28 are port 2 pins also referred to as P2.0 to P2.7. When additional external memory is interfaced with the 8051 microcontroller, pins of port 2 act as higher-order address bytes. These pins are bidirectional.

- **Pin 29 (PSEN) :**

PSEN stands for Program Store Enable. It is output, active-low pin. This is used to read external memory. In 8031 based system where external ROM holds the program code, this pin is connected to the OE pin of the ROM.

- **Pin 30 (ALE/ PROG) :**

ALE stands for Address Latch Enable. It is input, active-high pin. This pin is used to distinguish between memory chips when multiple memory chips are used. It is also used to de-multiplex the multiplexed address and data signals available at port 0.During flash programming i.e. Programming of EPROM, this pin acts as program pulse input (PROG).

- **Pin 31 (EA/ VPP) :**
  EA stands for External Access input. It is used to enable/disable external memory interfacing. In 8051, EA is connected to Vcc as it comes with on-chip ROM to store programs. For other family members such as 8031 and 8032 in which there is no on-chip ROM, the EA pin is connected to the GND.

- **Pin 32 to Pin 39 (Port 0) :**
  Pin 32 to pin 39 are port 0 pins also referred to as P0.0 to P0.7. They are bidirectional input/output pins. They don't have any internal pull-ups. Hence, 10K pull-up registers are used as external pull-ups. Port 0 is also designated as AD0- AD7 because 8051 multiplexes address anddata through port0 tosave pins.

- **Pin 40 (VCC) :**
  This pin provides power supply voltage i.e. +5 Volts to the circuit.

<u>Instruction Set of 8051:</u>

Types of Instructions in 8051 Microcontroller Instruction Set

Before seeing the types of instructions, let us see the structure of the 8051 Microcontroller Instruction. An 8051 Instruction consists of an Opcode (short of Operation – Code) followed by Operand(s) of size Zero Byte, One Byte or Two Bytes.

The Op-Code part of the instruction contains the Mnemonic, which specifies the type of operation to be performed. All Mnemonics or the Opcode part of the instruction are of One Byte size.

Coming to the Operand part of the instruction, it defines the data being processed by the instructions. The operand can be any of the following:

□ No Operand
□ Data value
□ I/O Port
□ Memory Location
□ CPU register

There can multiple operands and the format of instruction is as follows: MNEMONIC

DESTINATION OPERAND, SOURCE OPERAND

A simple instruction consists of just the opcode. Other instructions may include one or more operands. Instruction can be one-byte instruction, which contains only opcode, or two-byte instructions, where the second byte is the operand or three byte instructions, where the operand makes up the second and third byte.

Based on the operation they perform, all the instructions in the 8051 Microcontroller Instruction Set are divided into five groups. They are:

□ Data Transfer Instructions
□ Arithmetic Instructions
□ Logical Instructions
□ Boolean or Bit Manipulation Instructions
□ Program Branching Instructions

We will now see about these instructions briefly.

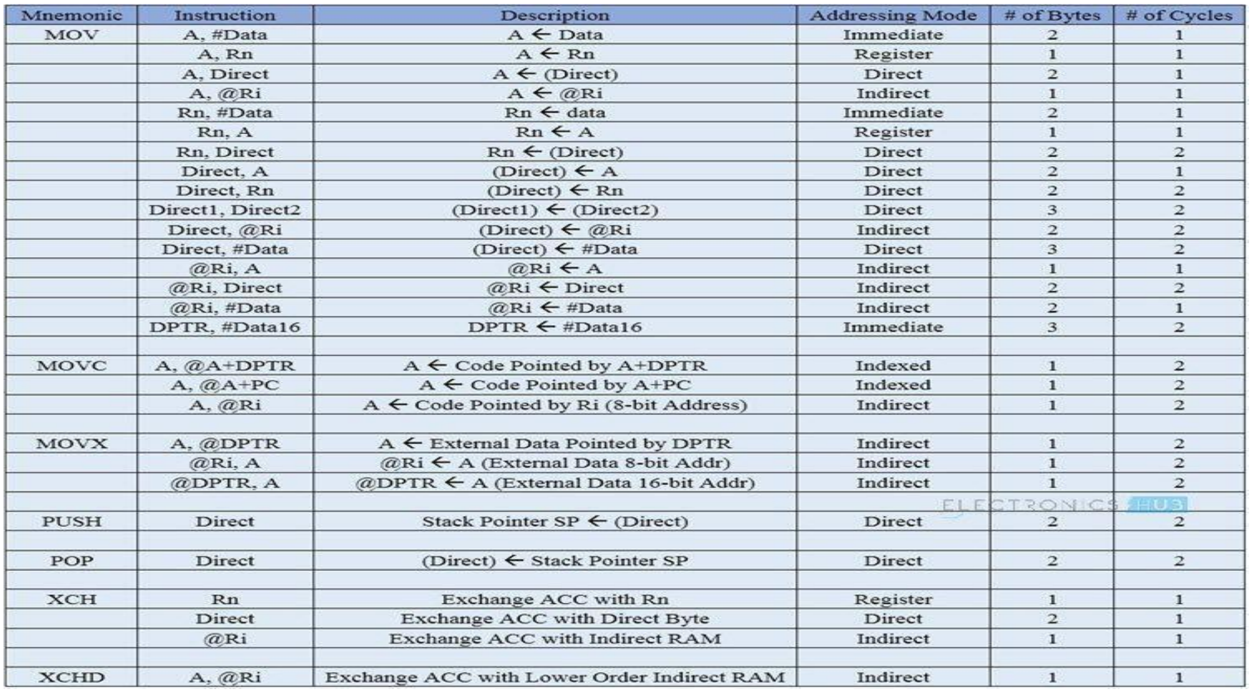
Data Transfer Instructions

The Data Transfer Instructions are associated with transfer of data between registers or external program memory or external data memory. The Mnemonics associated with Data Transfer are given below.

- *MOV*
- *MOVC*
- *MOVX*
- *PUSH*
- *POP*
- *XCH*
- *XCHD*

Mnemonic	**Description**
MOV	Move Data
MOVC	Move Code
MOCX	Move External Data
PUSH	Move Data to Stack
POP	Copy Data from Stack
XCH	Exchange Data between two Registers
XCHD	Exchange Lower Order Data between two Registers

The following table lists out all the possible data transfer instructions along with other details like addressing mode, size occupied and number machine cycles it takes.



Mnemonic	Instruction	Description	Addressing Mode	# of Bytes	# of Cycles
MOV	A, #Data	A ← Data	Immediate	2	1
	A, Rn	A ← Rn	Register	1	1
	A, Direct	A ← (Direct)	Direct	2	1
	A, @Ri	A ← @Ri	Indirect	1	1
	Rn, #Data	Rn ← data	Immediate	2	1
	Rn, A	Rn ← A	Register	1	1
	Rn, Direct	Rn ← (Direct)	Direct	2	2
	Direct, A	(Direct) ← A	Direct	2	1
	Direct, Rn	(Direct) ← Rn	Direct	2	2
	Direct1, Direct2	(Direct1) ← (Direct2)	Direct	3	2
	Direct, @Ri	(Direct) ← @Ri	Indirect	2	2
	Direct, #Data	(Direct) ← #Data	Direct	3	2
	@Ri, A	@Ri ← A	Indirect	1	1
	@Ri, Direct	@Ri ← Direct	Indirect	2	2
	@Ri, #Data	@Ri ← #Data	Indirect	2	1
	DPTR, #Data16	DPTR ← #Data16	Immediate	3	2
MOVC	A, @A+DPTR	A ← Code Pointed by A+DPTR	Indexed	1	2
	A, @A+PC	A ← Code Pointed by A+PC	Indexed	1	2
	A, @Ri	A ← Code Pointed by Ri (8-bit Address)	Indirect	1	2
MOVX	A, @DPTR	A ← External Data Pointed by DPTR	Indirect	1	2
	@Ri, A	@Ri ← A (External Data 8-bit Addr)	Indirect	1	2
	@DPTR, A	@DPTR ← A (External Data 16-bit Addr)	Indirect	1	2
PUSH	Direct	Stack Pointer SP ← (Direct)	Direct	2	2
POP	Direct	(Direct) ← Stack Pointer SP	Direct	2	2
XCH	Rn	Exchange ACC with Rn	Register	1	1
	Direct	Exchange ACC with Direct Byte	Direct	2	1
	@Ri	Exchange ACC with Indirect RAM	Indirect	1	1
XCHD	A, @Ri	Exchange ACC with Lower Order Indirect RAM	Indirect	1	1

Arithmetic Instructions:

Using Arithmetic Instructions, you can perform addition, subtraction, multiplication and division. The arithmetic instructions also include increment by one, decrement by one and a special instruction called Decimal Adjust Accumulator.

The Mnemonics associated with the Arithmetic Instructions of the 8051 Microcontroller Instruction Set are:

- *ADD*
- *ADDC*
- *SUBB*
- *INC*
- *DEC*
- *MUL*
- *DIV*
- *DA A*

Mnemonic	**Description**
ADD	Addition without Carry
ADDC	Addition with Carry
SUBB	Subtract with Carry
INC	Increment by 1
DEC	Decrement by 1
MUL	Multiply
DIV	Divide
DA A	Decimal Adjust the Accumulator (A Register)

The arithmetic instructions have no knowledge about the data format i.e., signed, unsigned, ASCII, BCD, etc. Also, the operations performed by the arithmetic instructions affect flags like carry, overflow, zero, etc. in the PSW Register.

All the possible Mnemonics associated with Arithmetic Instructions are mentioned in the following table.

Logical Instructions

The next group of instructions are the Logical Instructions, which perform logical operations like AND, OR, XOR, NOT, Rotate, Clear and Swap. Logical Instruction are performed on Bytes of data on a bit-by-bit basis.

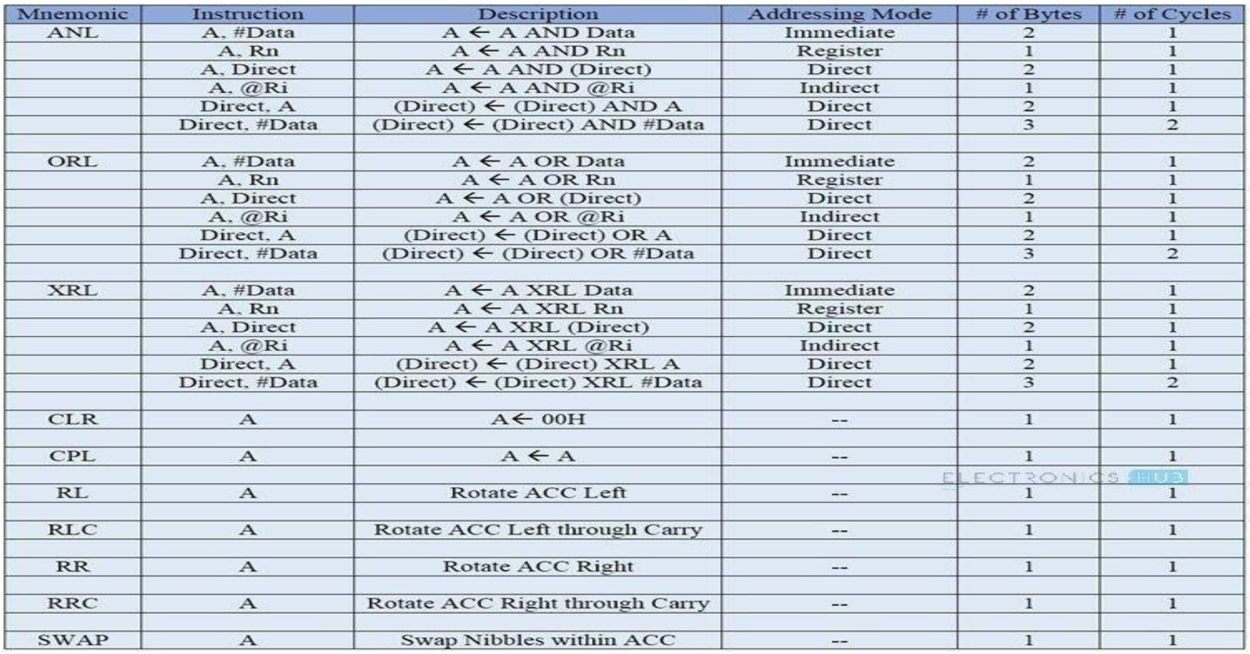
Mnemonics associated with Logical Instructions are as follows:

- *ANL*
- *ORL*
- *XRL*
- *CLR*
- *CPL*

- *RL*
- *RLC*
- *RR*
- *RRC*
- *SWAP*

Mnemonic	Description
ANL	Logical AND
ORL	Logical OR
XRL	Ex-OR
CLR	Clear Register
CPL	Complement the Register
RL	Rotate a Byte to Left
RLC	Rotate a Byte and Carry Bit to Left
RR	Rotate a Byte to Right
RRC	Rotate a Byte and Carry Bit to Right
SWAP	Exchange lower and higher nibbles in a Byte

The following table shows all the possible Mnemonics of the Logical Instructions.



Mnemonic	Instruction	Description	Addressing Mode	# of Bytes	# of Cycles
ANL	A, #Data	A ← A AND Data	Immediate	2	1
	A, Rn	A ← A AND Rn	Register	1	1
	A, Direct	A ← A AND (Direct)	Direct	2	1
	A, @Ri	A ← A AND @Ri	Indirect	1	1
	Direct, A	(Direct) ← (Direct) AND A	Direct	2	1
	Direct, #Data	(Direct) ← (Direct) AND #Data	Direct	3	2
ORL	A, #Data	A ← A OR Data	Immediate	2	1
	A, Rn	A ← A OR Rn	Register	1	1
	A, Direct	A ← A OR (Direct)	Direct	2	1
	A, @Ri	A ← A OR @Ri	Indirect	1	1
	Direct, A	(Direct) ← (Direct) OR A	Direct	2	1
	Direct, #Data	(Direct) ← (Direct) OR #Data	Direct	3	2
XRL	A, #Data	A ← A XRL Data	Immediate	2	1
	A, Rn	A ← A XRL Rn	Register	1	1
	A, Direct	A ← A XRL (Direct)	Direct	2	1
	A, @Ri	A ← A XRL @Ri	Indirect	1	1
	Direct, A	(Direct) ← (Direct) XRL A	Direct	2	1
	Direct, #Data	(Direct) ← (Direct) XRL #Data	Direct	3	2
CLR	A	A← 00H	--	1	1
CPL	A	A ← A	--	1	1
RL	A	Rotate ACC Left	--	1	1
RLC	A	Rotate ACC Left through Carry	--	1	1
RR	A	Rotate ACC Right	--	1	1
RRC	A	Rotate ACC Right through Carry	--	1	1
SWAP	A	Swap Nibbles within ACC	--	1	1

Boolean or Bit Manipulation Instructions

As the name suggests, Boolean or Bit Manipulation Instructions deal with bit variables. We know that there is a special bit-addressable area in the RAM and some of the Special Function Registers (SFRs) are also bit addressable.

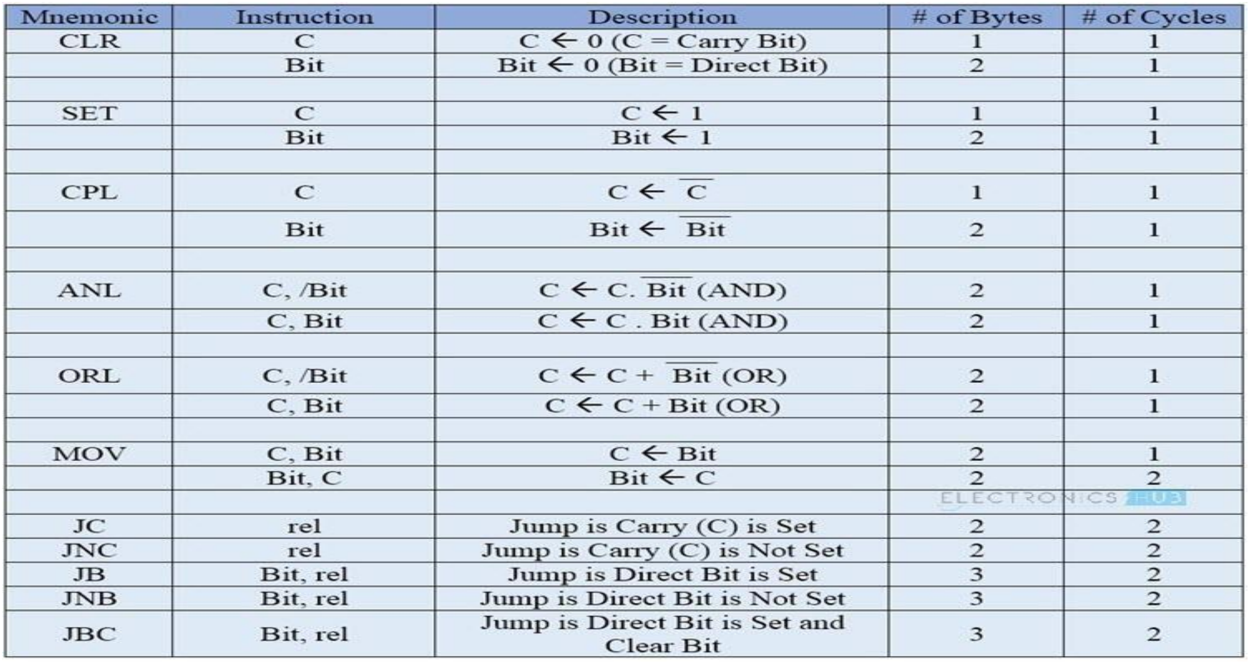
The Mnemonics corresponding to the Boolean or Bit Manipulation instructions are:

- *CLR*
- *SETB*

- *MOV*
- *JC*
- *JNC*
- *JB*
- *JNB*
- *JBC*
- *ANL*
- *ORL*
- *CPL*

Mnemonic	**Description**
CLR	Clear a Bit (Reset to 0)
SETB	Set a Bit (Set to 1)
MOV	Move a Bit
JC	Jump if Carry Flag is Set
JNC	Jump if Carry Flag is Not Set
JB	Jump if specified Bit is Set
JNB	Jump if specified Bit is Not Set
JBC	Jump if specified Bit is Set and also clear the Bit
ANL	Bitwise AND
ORL	Bitwise OR
CPL	Complement the Bit

These instructions can perform set, clear, and, or, complement etc. at bit level. All the possible mnemonics of the Boolean Instructions are specified in the following table.



Mnemonic	Instruction	Description	# of Bytes	# of Cycles
CLR	C	C ← 0 (C = Carry Bit)	1	1
	Bit	Bit ← 0 (Bit = Direct Bit)	2	1
SET	C	C ← 1	1	1
	Bit	Bit ← 1	2	1
CPL	C	C ← $\overline{C}$	1	1
	Bit	Bit ← $\overline{Bit}$	2	1
ANL	C, /Bit	C ← C. $\overline{Bit}$ (AND)	2	1
	C, Bit	C ← C . Bit (AND)	2	1
ORL	C, /Bit	C ← C + $\overline{Bit}$ (OR)	2	1
	C, Bit	C ← C + Bit (OR)	2	1
MOV	C, Bit	C ← Bit	2	1
	Bit, C	Bit ← C	2	2
JC	rel	Jump is Carry (C) is Set	2	2
JNC	rel	Jump is Carry (C) is Not Set	2	2
JB	Bit, rel	Jump is Direct Bit is Set	3	2
JNB	Bit, rel	Jump is Direct Bit is Not Set	3	2
JBC	Bit, rel	Jump is Direct Bit is Set and Clear Bit	3	2

Program Branching Instructions

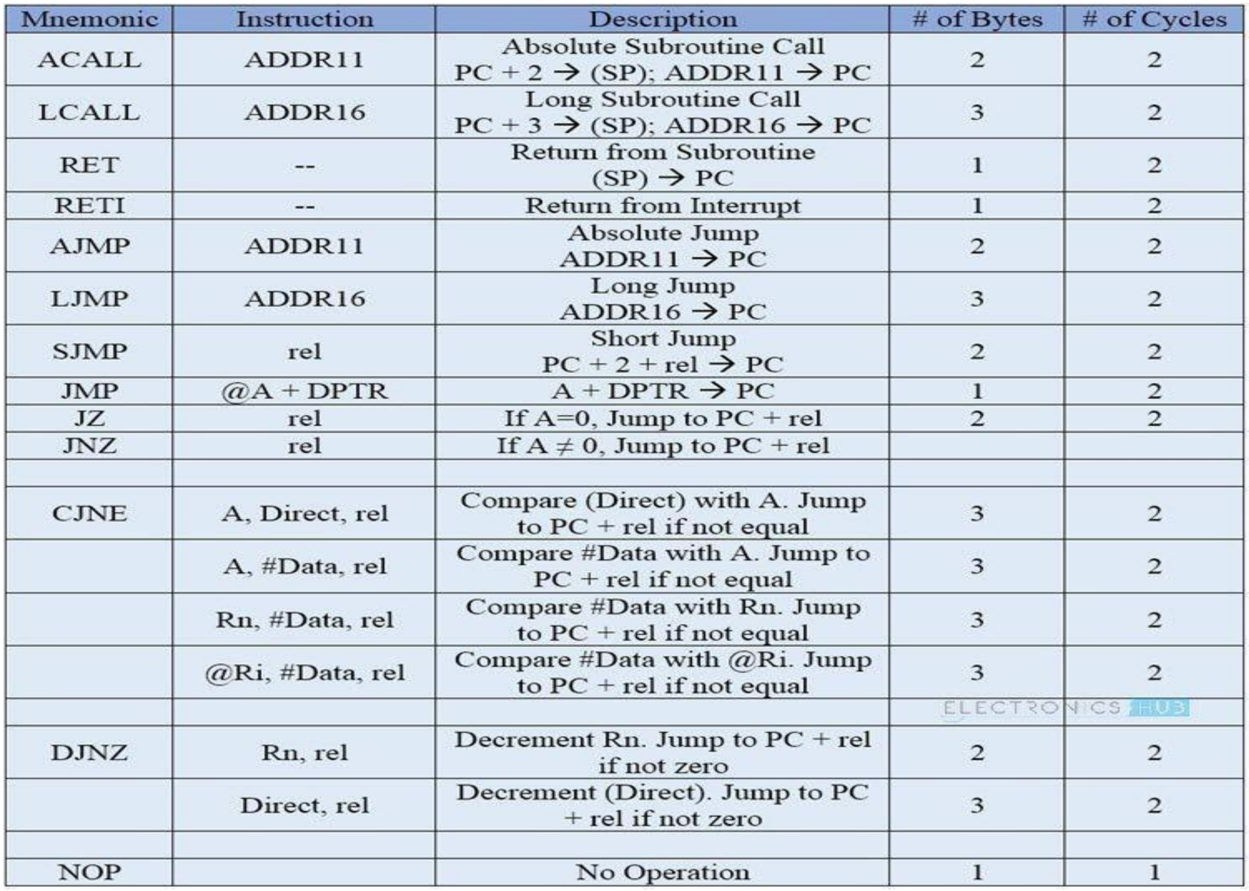
The last group of instructions in the 8051 Microcontroller Instruction Set are the Program Branching Instructions. These instructions control the flow of program logic. The mnemonics of the Program Branching Instructions are as follows.

- *LJMP*
- *AJMP*
- *SJMP*
- *JZ*
- *JNZ*
- *CJNE*
- *DJNZ*
- *NOP*
- *LCALL*
- *ACALL*
- *RET*
- *RETI*
- *JMP*

Mnemonic	**Description**
LJMP	Long Jump (Unconditional)
AJMP	Absolute Jump (Unconditional)
SJMP	Short Jump (Unconditional)
JZ	Jump if A is equal to 0
JNZ	Jump if A is not equal to 0
CJNE	Compare and Jump if Not Equal
DJNZ	Decrement and Jump if Not Zero
NOP	No Operation
LCALL	Long Call to Subroutine
ACALL	Absolute Call to Subroutine (Unconditional)
RET	Return from Subroutine
RETI	Return from Interrupt
JMP	Jump to an Address (Unconditional)

All these instructions, except the NOP (No Operation) affect the Program Counter (PC) in one way or other. Some of these instructions has decision making capability before transferring control to other part of the program.

The following table shows all the mnemonics with respect to the program branching instructions.



Mnemonic	Instruction	Description	# of Bytes	# of Cycles
ACALL	ADDR11	Absolute Subroutine Call PC + 2 → (SP); ADDR11 → PC	2	2
LCALL	ADDR16	Long Subroutine Call PC + 3 → (SP); ADDR16 → PC	3	2
RET	--	Return from Subroutine (SP) → PC	1	2
RETI	--	Return from Interrupt	1	2
AJMP	ADDR11	Absolute Jump ADDR11 → PC	2	2
LJMP	ADDR16	Long Jump ADDR16 → PC	3	2
SJMP	rel	Short Jump PC + 2 + rel → PC	2	2
JMP	@A + DPTR	A + DPTR → PC	1	2
JZ	rel	If A=0, Jump to PC + rel	2	2
JNZ	rel	If A ≠ 0, Jump to PC + rel		
CJNE	A, Direct, rel	Compare (Direct) with A. Jump to PC + rel if not equal	3	2
	A, #Data, rel	Compare #Data with A. Jump to PC + rel if not equal	3	2
	Rn, #Data, rel	Compare #Data with Rn. Jump to PC + rel if not equal	3	2
	@Ri, #Data, rel	Compare #Data with @Ri. Jump to PC + rel if not equal	3	2
DJNZ	Rn, rel	Decrement Rn. Jump to PC + rel if not zero	2	2
	Direct, rel	Decrement (Direct). Jump to PC + rel if not zero	3	2
NOP		No Operation	1	1

Microcontrollers 8051 Input Output Ports

8051 microcontrollers have 4 I/O ports each of 8-bit, which can be configured as input or output. Hence, total 32 input/output pins allow the microcontroller to be connected with the peripheral devices.

- **Pin configuration**, i.e. the pin can be configured as 1 for input and 0 for output as per the logic state.
  - **Input/Output (I/O) pin** − All the circuits within the microcontroller must be connected to one of its pins except P0 port because it does not have pull- up resistors built-in.
  - **Input pin** − Logic 1 is applied to a bit of the P register. The output FE transistor is turned off and the other pin remains connected to the power supply voltage over a pull-up resistor of high resistance.
- **Port 0** − The P0 (zero) port is characterized by two functions −
  - When the external memory is used then the lower address byte (addresses A0A7) is applied on it, else all bits of this port are configured as input/output.
  - When P0 port is configured as an output then other ports consisting of pins with built-in pull-up resistor connected by its end to 5V power supply, the pins of this port have this resistor left out.

Input Configuration:

If any pin of this port is configured as an input, then it acts as if it “floats”, i.e. the input has unlimited input resistance and in-determined potential.

Output Configuration:

When the pin is configured as an output, then it acts as an “open drain”. By applying logic 0 to a port bit, the appropriate pin will be connected to ground (0V), and applying logic 1, the external output will keep on “floating”.

In order to apply logic 1 (5V) on this output pin, it is necessary to build an external pullup resistor.

- Port 1

P1 is a true I/O port as it doesn’t have any alternative functions as in P0, but this port can be configured as general I/O only. It has a built-in pull-up resistor and is completely compatible with TTL circuits.

- Port 2

P2 is similar to P0 when the external memory is used. Pins of this port occupy addresses intended for the external memory chip. This port can be used for higher address byte with addresses A8-A15. When no memory is added then this port can be used as a general input/output port similar to Port 1.

- Port 3

In this port, functions are similar to other ports except that the logic 1 must be applied to appropriate bit of the P3 register.

Pins Current Limitations

□ When pins are configured as an output (i.e. logic 0), then the single port pins can receive a current of 10mA.

□ When these pins are configured as inputs (i.e. logic 1), then built-in pull-up resistors provide very weak current, but can activate up to 4 TTL inputs of LS series.

□ If all 8 bits of a port are active, then the total current must be limited to 15mA (port P0: 26mA).

□ If all ports (32 bits) are active, then the total maximum current must be limited to 71mA.